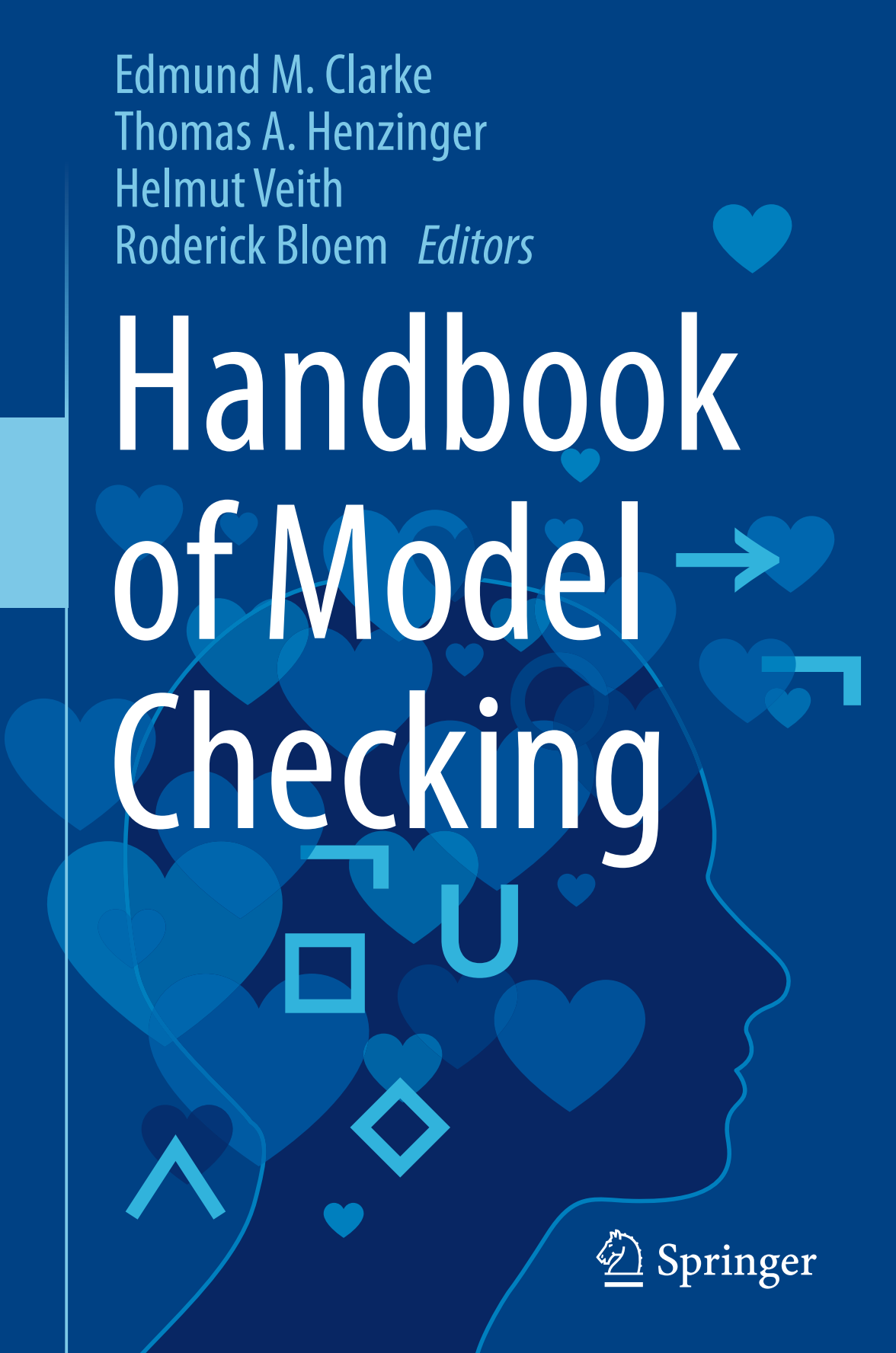
Edmund M. Clarke
Thomas A. Henzinger
Helmut Veith
Roderick Bloem *Editors*

Handbook of Model Checking



Springer

Handbook of Model Checking

Edmund M. Clarke • Thomas A. Henzinger •
Helmut Veith • Roderick Bloem
Editors

Handbook of Model Checking

Springer

Editors
Edmund M. Clarke
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA

Thomas A. Henzinger
Institute of Science and Technology Austria
(IST Austria)
Klosterneuburg, Austria

Helmut Veith
Arbeitsbereich Formal Methods in Systems
Engineering
Technische Universität Wien
Wien, Austria

Roderick Bloem
Institut für angewandte
Informationsverarbeitung und
Kommunikationstechnologie (IAIK)
Technische Universität Graz
Graz, Austria

ISBN 978-3-319-10574-1
DOI 10.1007/978-3-319-10575-8

ISBN 978-3-319-10575-8 (eBook)

Library of Congress Control Number: 2018943403

The image on the book cover was designed by Anna Petukhova

Printed on acid-free paper

This Springer imprint is published by the registered company
Springer International Publishing AG part of Springer Nature.
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

To Helmut

Foreword

With 32 technical articles and 76 authors, this handbook represents a full post-graduate course in Model Checking. If a reader can verify that he or she has read and studied every article, then Springer should certainly award that reader a Master's Degree in Model Checking! Departments in Computer Science everywhere will certainly welcome access to this major resource.

Model Checking has become a major area of research and development both for hardware and software verification owing to many factors. First, the improved speed and capacity of computers in recent times have made all kinds of problem solving both practical and efficient. Moreover, in the area of Model Checking the methods of design of models have contributed to the best formulation of problems. Then we have seen SAT solvers gain unexpected and truly remarkable efficiency improvements—despite theoretical limitations. Additionally, the methodology of Satisfiability Modulo Theories (SMT) has contributed to finding excellent ways to pose and solve problems. Uses of temporal logic and data-flow-analysis techniques have also made model checking more naturally efficient. All these contributions have helped solve the ever-present "state explosion problem." The urgency to make greater strides has increased because new applications in such diverse areas as health care, transportation, security, and robotics require work in the field to achieve greater scale, expressivity, and automation.

I would definitely recommend new Ph.D. candidates look seriously into going into research in this field, because success in Model Checking can directly lead to future success in many other activities in Computer Science.

Finally, the recent tragic loss of Helmut Veith has been a dreadful blow to his family, friends, colleagues, and students. Let's take up the flag in his honor to help promote and expand the field in which he was poised to become a recognized world leader.

Carnegie Mellon University
Department of Mathematics, University of California, Berkeley

Dana S. Scott

Preface

This handbook is intended to give an in-depth description of the many research areas that make up the expanding field of model checking. In 32 chapters, 76 of the world's leading researchers in this domain present a thorough review of the origins, theory, methods, and applications of model checking. The book is meant for researchers and graduate students who are interested in the development of formalisms, algorithms, and software tools for the computer-aided verification of complex systems in general, and of hardware and software systems in particular.

The idea for this handbook originated with Helmut Veith around 2006. It was clear to Helmut that a field as strong and useful as model checking needed a handbook to make its foundations broadly accessible. Helmut was in many ways the soul of this project. His untimely death in March 2016, with the project in its final phase, was a shock to all of us and we greatly miss him. His visionary ideas, his unbelievable energy in bringing these ideas to life, and his wonderful sense of community left a lasting mark, and this book will serve as an enduring memorial to his contributions to the field and the community.

Graz, Klosterneuburg, Pittsburgh
November 2016

Roderick Bloem
Thomas A. Henzinger
Edmund M. Clarke

Acknowledgements

Very special thanks go to Katarina Singer and Sasha Rubin. As project coordinators, they assisted the editors in all managerial, organizational, and technical matters necessary for bringing such a large project to fruition: they managed the collaboration software and the interaction with the authors, reviewers, copy editors, and the publisher throughout much of the project. They kept track of all the loose ends and spent many hours making sure that every chapter got into shape. Jessica Davies and Andrey Kupriyanov performed some of these coordination tasks for shorter periods of time towards the end of the project. Thank you!

We would like to thank Alfred Hofmann of Springer for his support of the project. Ronan Nugent, our editor at Springer, was closely involved in all aspects of the book from the beginning to its completion. He deserves special credit for flexibility, patience, and for keeping an unrelenting trust in the project. We also thank Philip R. Watson, Springer's copy editor, who labored through each and every chapter and found countless mistakes. The copy editing was supported by many volunteers, who proofread individual chapters, integrated changes suggested by Springer, discussed problems with the authors, and helped create the index. The editing team was largely based at IST Austria, TU Graz and TU Vienna and consisted of Guy Avni, Sergiy Bogomolov, Przemysław Daca, Jessica Davies, Masoud Ebrahimi, Thomas Ferrère, Miriam García Soto, Mirco Giacobbe, Vedad Hadzic, Rinat Iusupov, Barbara Jobstmann, Anja Karl, Ayrat Khalimov, Bettina Könighofer, Hui Kong, Igor Konnov, Tomer Kotek, Bernhard Kragl, Jan Křetínský, Orna Kupferman, Andrey Kupriyanov, Alfons Laarman, Kurt Nistelberger, Jan Otop, Leo Prikler, Heinz Riener, Franz Röck, Sasha Rubin, Roopsha Samanta, Moritz Sinn, Ana Sokolova, Thorsten Tarrach, Ursula Urwanisch, and Georg Weissenbacher. Thank you!

A large group of people reviewed all of the chapters, often in great detail. We are grateful to Parosh Aziz Abdulla, Rajeev Alur, Domagoj Babic, Christel Baier, Clark Barrett, David Basin, Jason Baumgartner, Saddek Bensalem, Dietmar Berwanger, Dirk Beyer, Armin Biere, Bernard Boigelot, Marius Bozga, Julian Bradfield, Robert K. Brayton, Tomáš Brázdil, Randal E. Bryant, Pavol Černy, Sagar Chaki, Krishnendu Chatterjee, Swarat Chaudhuri, Wei-Ngan Chin, Hana Chockler,

Alessandro Cimatti, Rance Cleaveland, Cas Cremers, Dennis Dams, Thao Dang, Leonardo de Moura, David Dill, Laurent Doyen, Cezara Drăgoi, Rolf Drechsler, Matt Dwyer, Cindy Eisner, Javier Esparza, Kousha Etessami, Marco Faella, Azadeh Farzan, Bernd Finkbeiner, Alain Finkel, Dana Fisman, Vojtěch Forejt, Goran Frehse, Masahiro Fujita, Sicun Gao, Patrice Godefroid, Ganesh Gopalakrishnan, Erich Grädel, Alex Groce, Radu Grosu, Orna Grumberg, Sumit Gulwani, Aarti Gupta, Arie Gurfinkel, Reiner Hähnle, John Hatcliff, Keijo Heljanko, David Henriques, Holger Hermanns, Kryštof Hoder, Martin Hofmann, Andreas Holzer, Gerard Holzmann, Radu Iosif, Franjo Ivančić, Ranjit Jhala, Barbara Jobstmann, Joost-Pieter Katoen, Shmuel Katz, Stefan Katzenbeisser, Stefan Kiefer, Johannes Kinder, Felix Klaedtke, Igor Konnov, Eric Koskinen, Laura Kovács, Daniel Kroening, Antonín Kučera, Andreas Kuehlmann, Viktor Kunčak, Orna Kupferman, Robert P. Kurshan, Marta Kwiatowska, Shuvendu Lahiri, Yassine Lakhnech, Akash Lal, Salvatore La Torre, Martin Leucker, Rupak Majumdar, Oded Maler, Tiziana Margaria, Nicolas Markey, Joao Marquez-Silva, Wilfredo Marrero, João Martins, Richard Mayr, Catherine Meadows, Tom Melham, Shin-ichi Minato, Marius Minea, David Monniaux, Kedar S. Namjoshi, Dejan Ničković, Joël Ouaknine, Madhusudan Parthasarathy, Corina Păsăreanu, Doron Peled, Ruzica Piskac, André Platzer, Stefano Quer, Jan-David Quesel, Jean-François Raskin, Tom Reps, Sasha Rubin, John Rushby, Andrey Rybalchenko, Mooly Sagiv, Sriram Sankaranarayanan, Christian Schallhart, Sven Schewe, Holger Schlingloff, David A. Schmidt, Philippe Schnoebelen, Stefan Schwoon, Helmut Seidl, Martina Seidl, Koushik Sen, Sanjit A. Seshia, Natarajan Shankar, Natasha Sharygina, Nishant Sinha, Prasad Sistla, Ana Sokolova, Dawn Song, Bernhard Steffen, Wilfried Steiner, Andreas Steininger, Mariëlle Stoelinga, Ofer Strichman, Aaron Stump, Stefan Szeider, Murali Talupur, Wolfgang Thomas, Cesare Tinelli, Ashish Tiwari, Stavros Tripakis, Rachel Tzoref, Viktor Vafeiadis, Allen Van Gelder, Martin Vechev, Thomas Wahl, Igor Walukiewicz, Bow-Yaw Wang, Farn Wang, Georg Weissenbacher, Thomas Wies, Karsten Wolf, Pierre Wolper, Eran Yahav, Karen Yorav, Greta Yorsh, Vladimir Zakharov, and Paolo Zuliani.

The greatest debt of gratitude we owe to the authors, who did the most important job: writing the book. The editors were supported in the initial selection of topics and authors by an advisory board consisting of Rajeev Alur, Armin Biere, Robert K. Brayton, Randal E. Bryant, Rance Cleaveland, Cindy Eisner, Javier Esparza, Orna Grumberg, Aarti Gupta, Pei-Hsin Ho, Gerard Holzmann, Orna Kupferman, Robert P. Kurshan, Kim G. Larsen, Ken McMillan, Doron Peled, Amir Pnueli, Sriram Rajamani, Joseph Sifakis, Prasad Sistla, Scott Smolka, Wolfgang Thomas, and Pierre Wolper. Many of them contributed also to individual chapters. Finally, we thank Dana Scott for writing the Foreword, Andrei Voronkov for letting us use his EasyChair software for the initial writing and reviewing process, and Anna Petukhova for the design of the cover image, which is based on her design for the 2014 Vienna Summer of Logic.

Roderick Bloem
Thomas A. Henzinger
Edmund M. Clarke

Contents

1 Introduction to Model Checking 1
Edmund M. Clarke, Thomas A. Henzinger, and Helmut Veith
1.1 The Case for Computer-Aided Verification 1
1.2 Temporal-Logic Model Checking in a Nutshell 6
1.3 A Very Brief Guide Through the Chapters of the Handbook . . 13
1.4 The Future of Model Checking 19
References 22

2 Temporal Logic and Fair Discrete Systems 27
Nir Piterman and Amir Pnueli
2.1 Introduction 28
2.2 Fair Discrete Systems 29
2.3 Linear Temporal Logic 41
2.4 Computation Tree Logic 52
2.5 Examples for LTL and CTL 59
2.6 CTL* 63
References 70

3 Modeling for Verification 75
Sanjit A. Seshia, Natasha Sharygina, and Stavros Tripakis
3.1 Introduction 75
3.2 Major Considerations in System Modeling 77
3.3 Modeling Basics 86
3.4 Examples 89
3.5 Kripke Structures 100
3.6 Summary 103
References 103

4 Automata Theory and Model Checking 107
Orna Kupferman
4.1 Introduction 107
4.2 Nondeterministic Büchi Automata on Infinite Words 108

4.3 Additional Acceptance Conditions . . . 122
4.4 Decision Procedures . . . 132
4.5 Alternating Automata on Infinite Words . . . 135
4.6 Automata-Based Algorithms . . . 141
References . . . 148

5 Explicit-State Model Checking . . . 153
Gerard J. Holzmann
5.1 Introduction . . . 153
5.2 Basic Search Algorithms . . . 155
5.3 Linear Temporal Logic . . . 158
5.4 Omega Automata . . . 158
5.5 Nested Depth-First Search . . . 160
5.6 Abstraction . . . 162
5.7 Model-Driven Verification . . . 167
5.8 Incomplete Storage . . . 168
5.9 Extensions . . . 169
5.10 Synopsis . . . 170
References . . . 170

6 Partial-Order Reduction . . . 173
Doron Peled
6.1 Introduction . . . 173
6.2 Partial Order Reduction . . . 174
6.3 Reducing Edges While Preserving States . . . 182
6.4 Conclusions . . . 188
References . . . 188

7 Binary Decision Diagrams . . . 191
Randal E. Bryant
7.1 Introduction . . . 191
7.2 Terminology . . . 192
7.3 A Boolean Function API . . . 193
7.4 OBDD Representation . . . 195
7.5 Implementing OBDD Operations . . . 197
7.6 Implementation Techniques . . . 200
7.7 Variable Ordering and Reordering . . . 202
7.8 Variant Representations . . . 203
7.9 Representing Non-Boolean Functions . . . 206
7.10 Scaling OBDD Capacity . . . 210
7.11 Concluding Remarks . . . 213
References . . . 214

8 BDD-Based Symbolic Model Checking . . . 219
Sagar Chaki and Arie Gurfinkel
8.1 Introduction . . . 219
8.2 Preliminaries . . . 220

8.3 Binary Decision Diagrams: The Basics . . . 222
8.4 Model Checking Kripke Structures . . . 230
8.5 Push-Down Symbolic Model Checking . . . 238
8.6 Conclusion . . . 243
References . . . 244

9 Propositional SAT Solving . . . 247
Joao Marques-Silva and Sharad Malik
9.1 Introduction . . . 247
9.2 Preliminaries . . . 249
9.3 CDCL SAT Solvers: Organization . . . 252
9.4 CDCL SAT Solvers . . . 253
9.5 SAT-Based Problem Solving . . . 264
9.6 Research Directions . . . 269
References . . . 269

10 SAT-Based Model Checking . . . 277
Armin Biere and Daniel Kröning
10.1 Introduction . . . 277
10.2 Bounded Model Checking on Kripke Structures . . . 278
10.3 Bounded Model Checking for Hardware Designs . . . 281
10.4 Bounded Model Checking for Software . . . 283
10.5 Encodings into Propositional SAT . . . 287
10.6 Complete Model Checking with SAT . . . 289
10.7 Abstraction Techniques Using SAT . . . 292
10.8 Outlook and Conclusions . . . 295
References . . . 295

11 Satisfiability Modulo Theories . . . 305
Clark Barrett and Cesare Tinelli
11.1 Introduction . . . 305
11.2 SMT in Model Checking . . . 310
11.3 The Lazy Approach to SMT . . . 312
11.4 Theory Solvers for Specific Theories . . . 317
11.5 Combining Theory Solvers . . . 324
11.6 SMT Solving Extensions and Enhancements . . . 327
11.7 Eager Encodings to SAT . . . 330
11.8 Additional Functionalities of SMT Solvers . . . 332
References . . . 335

12 Compositional Reasoning . . . 345
Dimitra Giannakopoulou, Kedar S. Namjoshi, and Corina S. Păsăreanu
12.1 Introduction . . . 345
12.2 Reasoning with Assertions . . . 348
12.3 Automata-Based Assume-Guarantee Reasoning . . . 362
12.4 Related Approaches . . . 375
12.5 Conclusion . . . 378
References . . . 378

13 Abstraction and Abstraction Refinement . . . 385
Dennis Dams and Orna Grumberg
13.1 Introduction . . . 385
13.2 Preliminaries . . . 387
13.3 Simulation and Bisimulation Relations . . . 394
13.4 Abstraction Based on Simulation . . . 399
13.5 CounterExample-Guided Abstraction Refinement (CEGAR) . . . 402
13.6 Abstraction Based on Modal Simulation . . . 406
13.7 Completeness . . . 412
References . . . 414

14 Interpolation and Model Checking . . . 421
Kenneth L. McMillan
14.1 Introduction . . . 421
14.2 Preliminaries . . . 423
14.3 Model of Abstraction Refinement . . . 424
14.4 Refinement, Local Proofs, and Interpolants . . . 428
14.5 Refiners as Local Proof Systems . . . 434
14.6 Abstractors as Proof Generalizers . . . 441
14.7 Summary . . . 443
References . . . 444

15 Predicate Abstraction for Program Verification . . . 447
Ranjit Jhala, Andreas Podelski, and Andrey Rybalchenko
15.1 Introduction . . . 447
15.2 Definitions . . . 448
15.3 Characterizing Correctness via Reachability . . . 452
15.4 Characterizing Correctness via Inductiveness . . . 454
15.5 Abstraction . . . 459
15.6 Abstraction Refinement . . . 471
15.7 Solving Refinement Constraints for Predicate Abstraction . . . 481
15.8 Tools . . . 486
15.9 Conclusion . . . 487
References . . . 487

16 Combining Model Checking and Data-Flow Analysis . . . 493
Dirk Beyer, Sumit Gulwani, and David A. Schmidt
16.1 Introduction . . . 494
16.2 General Considerations . . . 494
16.3 Unifying Formal Framework/Comparison of Algorithms . . . 501
16.4 Classic Examples (Component Analyses) . . . 511
16.5 Combination Examples (Composite Analyses) . . . 520
16.6 Algorithms for Constructing Program Invariants . . . 525
16.7 Combinations in Tool Implementations . . . 532
16.8 Conclusion . . . 532
References . . . 534

17 Model Checking Procedural Programs 541
Rajeev Alur, Ahmed Bouajjani, and Javier Esparza
17.1 Introduction 541
17.2 Models of Procedural Programs 543
17.3 Basic Verification Algorithms 547
17.4 Specifying Requirements 556
17.5 Bibliographical Remarks 566
References 569

18 Model Checking Concurrent Programs 573
Aarti Gupta, Vineet Kahlon, Shaz Qadeer, and Tayssir Touili
18.1 Introduction 574
18.2 Concurrent System Model and Notation 576
18.3 PDS-Based Model Checking: Synchronization Patterns 580
18.4 PDS-Based Model-Checking: Communication Patterns 595
18.5 Other Models: Finite State Systems and Sequential Programs . 602
References 607

19 Combining Model Checking and Testing 613
Patrice Godefroid and Koushik Sen
19.1 Introduction 613
19.2 Systematic Testing of Concurrent Software 615
19.3 Systematic Testing of Sequential Software 624
19.4 Systematic Testing of Concurrent Software with Data Inputs . . 633
19.5 Other Related Work 637
19.6 Conclusion 640
References 640

20 Combining Model Checking and Deduction 651
Natarajan Shankar
20.1 Introduction 651
20.2 Logic Background 656
20.3 Deduction and Model Checking 670
20.4 Conclusions 680
References 680

21 Model Checking Parameterized Systems 685
Parosh Aziz Abdulla, A. Prasad Sistla, and Muralidhar Talupur
21.1 Introduction 685
21.2 Petri Nets 687
21.3 Regular Model Checking 695
21.4 Monotonic Abstraction 703
21.5 Compositional Reasoning for Parameterized Verification 709
21.6 Related Work 719
References 721

22 Model Checking Security Protocols . . . 727
David Basin, Cas Cremers, and Catherine Meadows
22.1 Introduction . . . 727
22.2 History . . . 731
22.3 Formal Model . . . 733
22.4 Issues in Developing Model-Checking Algorithms for Security Protocols . . . 741
22.5 Systems and Algorithms . . . 748
22.6 Research Problems . . . 753
22.7 Conclusions . . . 757
References . . . 758

23 Transfer of Model Checking to Industrial Practice . . . 763
Robert P. Kurshan
23.1 Introduction . . . 763
23.2 The Technology Transfer Problem . . . 767
23.3 False Starts . . . 776
23.4 A Framework for Technology Transfer . . . 779
23.5 Formal Functional Verification in Commercial Use Today . . . 782
23.6 Algorithms . . . 786
23.7 Future . . . 788
23.8 Conclusion . . . 789
References . . . 790

24 Functional Specification of Hardware via Temporal Logic . . . 795
Cindy Eisner and Dana Fisman
24.1 Introduction . . . 795
24.2 From LTL to Regular-Expression-Based Temporal Logic . . . 797
24.3 Clocks and Sampling . . . 805
24.4 Hardware Resets and Other Sources of Truncated Paths . . . 809
24.5 The Simple Subset . . . 817
24.6 Quantified and Local Variables . . . 818
24.7 Summary and Open Issues . . . 823
References . . . 825

25 Symbolic Trajectory Evaluation . . . 831
Tom Melham
25.1 Introduction . . . 831
25.2 Notational Preliminaries . . . 833
25.3 Sequential Circuit Models in STE . . . 834
25.4 Trajectory Evaluation Logic . . . 838
25.5 The Fundamental Theorem of Trajectory Evaluation . . . 843
25.6 STE Model Checking . . . 844
25.7 Abstraction and Symbolic Indexing . . . 852
25.8 Compositional Reasoning . . . 857
25.9 GSTE and Other Extensions . . . 862

25.10 Summary and Prospects . . . 865
References . . . 867

26 The mu-calculus and Model Checking . . . 871
Julian Bradfield and Igor Walukiewicz
26.1 Introduction . . . 871
26.2 Basics . . . 872
26.3 Fundamental Properties . . . 890
26.4 Relations with Other Logics . . . 904
26.5 Related Work . . . 912
References . . . 914

27 Graph Games and Reactive Synthesis . . . 921
Roderick Bloem, Krishnendu Chatterjee, and Barbara Jobstmann
27.1 Introduction . . . 921
27.2 Theory of Graph-Based Games . . . 923
27.3 Reactive Synthesis . . . 936
27.4 Related Topics . . . 953
References . . . 954

28 Model Checking Probabilistic Systems . . . 963
Christel Baier, Luca de Alfaro, Vojtěch Forejt, and Marta Kwiatkowska
28.1 Introduction . . . 964
28.2 Modelling Probabilistic Concurrent Systems . . . 966
28.3 Probabilistic Computation Tree Logic . . . 975
28.4 Model-Checking Algorithms for MDPs and PCTL . . . 980
28.5 Linear Temporal Logic . . . 985
28.6 Model-Checking Algorithms for MDPs and LTL . . . 987
28.7 Tools, Applications and Model Construction . . . 990
28.8 Extensions of the Model and Specification Notations . . . 992
28.9 Conclusion . . . 993
References . . . 993

29 Model Checking Real-Time Systems . . . 1001
Patricia Bouyer, Uli Fahrenberg, Kim Guldstrand Larsen,
Nicolas Markey, Joël Ouaknine, and James Worrell
29.1 Introduction . . . 1001
29.2 Timed Automata . . . 1003
29.3 Checking Reachability . . . 1007
29.4 (Bi)simulation Checking . . . 1010
29.5 Language-Theoretic Properties . . . 1012
29.6 Timed Temporal Logics . . . 1018
29.7 Symbolic Algorithms, Data Structures, Tools . . . 1023
29.8 Weighted Timed Automata . . . 1028
29.9 Timed Games . . . 1034
29.10 Ongoing and Future Challenges . . . 1037
References . . . 1037

30 Verification of Hybrid Systems 1047
Laurent Doyen, Goran Frehse, George J. Pappas, and André Platzer
30.1 Introduction 1048
30.2 Basic Definitions 1049
30.3 Decidability and Undecidability Results 1055
30.4 Set-Based Reachability Analysis 1058
30.5 Abstraction-Based Verification 1075
30.6 Logic-Based Verification 1084
30.7 Verification Tools 1097
References 1102

31 Symbolic Model Checking in Non-Boolean Domains 1111
Rupak Majumdar and Jean-François Raskin
31.1 Introduction 1111
31.2 Transition Systems and Symbolic Verification 1112
31.3 Examples of Symbolic Verification 1119
31.4 Games and Symbolic Synthesis 1131
31.5 Probabilistic Systems 1137
31.6 Conclusion 1141
References 1143

32 Process Algebra and Model Checking 1149
Rance Cleaveland, A.W. Roscoe, and Scott A. Smolka
32.1 Introduction 1149
32.2 Foundations 1150
32.3 Algorithms and Methodologies 1175
32.4 Tools 1181
32.5 Case Studies 1185
32.6 Conclusions 1190
References 1191

Index 1197

Contributors

Parosh Aziz Abdulla Uppsala University, Uppsala, Sweden

Rajeev Alur University of Pennsylvania, Philadelphia, PA, USA

Christel Baier Technische Universität Dresden, Dresden, Germany

Clark Barrett Stanford University, Stanford, CA, USA

David Basin ETH Zürich, Zürich, Switzerland

Dirk Beyer Ludwig-Maximilians-Universität München, Munich, Germany

Armin Biere Johannes Kepler University, Linz, Austria

Roderick Bloem Graz University of Technology, Graz, Austria

Ahmed Bouajjani Institut Universitaire de France, Paris Diderot University (Paris 7), Paris, France

Patricia Bouyer LSV, CNRS & ENS Paris-Saclay, Cachan, France

Julian Bradfield University of Edinburgh, Edinburgh, UK

Randal E. Bryant Carnegie Mellon University, Pittsburgh, PA, USA

Sagar Chaki Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA

Krishnendu Chatterjee IST Austria, Klosterneuburg, Austria

Edmund M. Clarke Carnegie Mellon University, Pittsburgh, PA, USA

Rance Cleaveland University of Maryland, College Park, College Park, MD, USA

Cas Cremers University of Oxford, Oxford, UK

Dennis Dams Embedded Systems Innovation by TNO, Eindhoven, The Netherlands

Luca de Alfaro University of California, Santa Cruz, Santa Cruz, CA, USA

Laurent Doyen LSV, CNRS & ENS Paris-Saclay, Cachan, France

Cindy Eisner IBM Research – Haifa, Haifa, Israel

Javier Esparza Technical University of Munich, Munich, Germany

Uli Fahrenberg École Polytechnique, Palaiseau, France

Dana Fisman Ben-Gurion University, Be'er Sheva, Israel

Vojtěch Forejt University of Oxford, Oxford, UK

Goran Frehse Verimag, University Grenoble Alpes, Grenoble, France

Dimitra Giannakopoulou NASA Ames Research Center, Moffett Field, CA, USA

Patrice Godefroid Microsoft Research, Redmond, WA, USA

Orna Grumberg Technion, Israel Institute of Technology, Haifa, Israel

Sumit Gulwani Microsoft Research, Redmond, WA, USA

Aarti Gupta Princeton University, Princeton, NJ, USA

Arie Gurfinkel University of Waterloo, Waterloo, ON, Canada

Thomas A. Henzinger IST Austria, Klosterneuburg, Austria

Gerard J. Holzmann Nimble Research, Monrovia, CA, USA

Ranjit Jhala Jacobs School of Engineering, University of California, San Diego, San Diego, CA, USA

Barbara Jobstmann École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland

Vineet Kahlon Google Inc., New York, NY, USA

Daniel Kröning University of Oxford, Oxford, UK

Orna Kupferman Hebrew University, Jerusalem, Israel

Robert P. Kurshan New York, NY, USA

Marta Kwiatkowska University of Oxford, Oxford, UK

Kim Guldstrand Larsen Aalborg University, Aalborg, Denmark

Rupak Majumdar Max Planck Institute for Software Systems, Kaiserslautern, Germany

Sharad Malik Princeton University, Princeton, NJ, USA

Nicolas Markey LSV, CNRS & ENS Paris-Saclay, Cachan, France; IRISA, CNRS & INRIA & Univ. Rennes 1, Rennes, France

Joao Marques-Silva University of Lisbon, Lisbon, Portugal

Kenneth L. McMillan Microsoft Research, Redmond, WA, USA

Catherine Meadows Naval Research Laboratory, Washington, DC, USA

Tom Melham University of Oxford, Oxford, UK

Kedar S. Namjoshi Bell Labs, Nokia, Murray Hill, NJ, USA

Joël Ouaknine University of Oxford, Oxford, UK; Max Planck Institute for Software Systems, Saarbrücken, Germany

George J. Pappas University of Pennsylvania, Philadelphia, PA, USA

Corina S. Păsăreanu NASA Ames Research Center, Moffett Field, CA, USA

Doron Peled Bar Ilan University, Ramat Gan, Israel

Nir Piterman University of Leicester, Leicester, UK

André Platzer Carnegie Mellon University, Pittsburgh, PA, USA

Amir Pnueli New York University, New York, USA; The Weizmann Institute of Science, Rehovot, Israel

Andreas Podelski University of Freiburg, Freiburg, Germany

Shaz Qadeer Microsoft Research, Redmond, WA, USA

Jean-François Raskin Université libre de Bruxelles, Brussels, Belgium

A.W. Roscoe University of Oxford, Oxford, UK

Andrey Rybalchenko Microsoft Research, Cambridge, UK

David A. Schmidt Kansas State University, Manhattan, KS, USA

Koushik Sen University of California, Berkeley, Berkeley, CA, USA

Sanjit A. Seshia University of California, Berkeley, Berkeley, CA, USA

Natarajan Shankar SRI International, Menlo Park, CA, USA

Natasha Sharygina Università della Svizzera italiana, Lugano, Switzerland

A. Prasad Sistla University of Illinois, Chicago, IL, USA

Scott A. Smolka Stony Brook University, Stony Brook, NY, USA

Muralidhar Talupur Intel's Strategic CAD Labs, Hillsboro, OR, USA

Cesare Tinelli The University of Iowa, Iowa City, IA, USA

Tayssir Touili CNRS, Paris, France

Stavros Tripakis University of California, Berkeley, Berkeley, CA, USA; Aalto University, Espoo, Finland

Helmut Veith Technische Universität Wien, Vienna, Austria

Igor Walukiewicz CNRS, University of Bordeaux, Bordeaux, France

James Worrell University of Oxford, Oxford, UK

Chapter 1
Introduction to Model Checking

Edmund M. Clarke, Thomas A. Henzinger, and Helmut Veith

Abstract Model checking is a computer-assisted method for the analysis of dynamical systems that can be modeled by state-transition systems. Drawing from research traditions in mathematical logic, programming languages, hardware design, and theoretical computer science, model checking is now widely used for the verification of hardware and software in industry. This chapter is an introduction and short survey of model checking. The chapter aims to motivate and link the individual chapters of the handbook, and to provide context for readers who are not familiar with model checking.

1.1 The Case for Computer-Aided Verification

> *The only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness. [32]*

In the ideal world of Dijkstra's Turing Award Lecture 1972, programs are intellectually manageable, and every program grows hand in hand with a mathematical proof of the program's correctness. The history of computer science has proven Dijkstra's vision limited. Manual proofs, if at all, can be found only in students' exercises, research papers on algorithms, and certain critical application areas. Although the work of McCarthy, Floyd, Hoare, and other pioneers [7, 36, 45, 62, 66] provided us with formal proof systems for program correctness, little use is made of these techniques in practice. The main challenge is scalability: real-world software systems not only include complex control and data structures, but depend on much "context" such as libraries and interfaces to other code, including lower-level systems code. As

E.M. Clarke
Carnegie Mellon University, Pittsburgh, PA, USA

T.A. Henzinger (✉)
IST Austria, Klosterneuburg, Austria
e-mail: tah@ist.ac.at

H. Veith
Technische Universität Wien, Vienna, Austria

E.M. Clarke et al. (eds.), *Handbook of Model Checking*,
DOI 10.1007/978-3-319-10575-8_1

a result, proving a software system correct requires much more effort, knowledge, training, and ingenuity than writing the software in trial-and-error style. This asymmetry between coding and verification is the main motivation for computer-aided verification, i.e., the use of computers for the verification of software and hardware.

From a 1972 perspective, the computer industry has changed the world beyond recognition. Computer programs today have millions of lines of code, they are written and maintained by globally distributed teams over decades, and they are used in diverse and complex computing environments from micro-code to cloud computing. Computer science has become pervasive in production, transportation, infrastructure, health care, science, finance, administration, defense, and entertainment. Programs are the most complex machines built by humans, and have huge responsibilities for human safety, security, health, and well-being. These developments have exacerbated the challenges and, at the same time, dramatically increased the need for correct programs and, hence, for computer-aided verification.

Starting with the work of Turing, the perspectives for automated verification did not look promising. Turing's halting problem [82] and Rice's Theorem [79] tell us that computer-aided verification is, in general, an unsolvable problem. At face value, these theorems demonstrate the undecidability of verification even for simple properties of simple programs. Technically, all that is needed for undecidability are two integer variables that, embedded into a looping control structure, can be incremented, decremented, and checked for zero. If the values of integer variables are bounded, we obtain a system with finitely many different states, and verification becomes decidable. However, complexity theory tells us that even for finite-state systems, many verification questions require a prohibitive effort.

Yet, at a time when logic in computer science was a synonym for undecidability and intractability, the invention of model checking marked a paradigm shift towards the practical use of logic for bug finding—i.e., falsification rather than verification—in the hardware and software industries. Not untypical of paradigms acquired many decades ago, the case for model checking appears simple and convincing in retrospect [22, 23, 72, 75]. In its basic classical form, the paradigm consists of the following insights:

Modeling. Finite state-transition graphs provide an adequate formalism for the description of finite-state systems such as hardware, but also for finite-state abstractions of software and of communication protocols.

Specification. Temporal logics provide a natural framework for the description of correctness properties for state-transition systems.

Algorithms. There are decision procedures for determining whether a finite state-transition structure is a model of a temporal-logic formula. Moreover, the decision procedures can produce diagnostic counterexamples when the formula is not true in the structure.

Taken together, these insights motivate the methodology that is shown in Fig. 1: the system under investigation is compiled into a state-transition graph (a.k.a. Kripke structure) K, the specification is expressed as a temporal-logic formula φ, and a decision procedure—the model checker—decides whether $K \models \varphi$, i.e., whether the

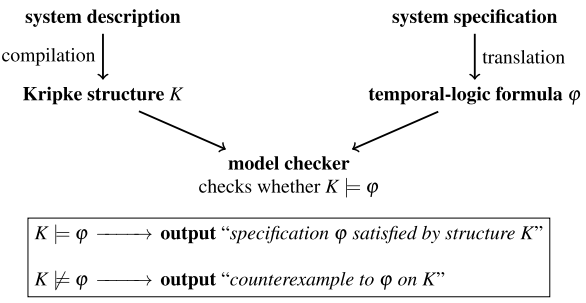


Fig. 1 Basic model-checking methodology

structure K is a model of the formula φ. If $K \not\models \varphi$, then the model checker outputs a counterexample that witnesses the violation of φ by K. The generation of counterexamples means that, in practice, falsification (the detection of bugs) can often be faster than verification (the proof of their absence).

There is of course a mismatch between the early model checkers, which were essentially graph algorithms on Kripke structures, and the complexity of modern computer systems sketched above. Research in model checking spanning more than three decades has helped to close this gap in many areas that are documented throughout this handbook. We can roughly classify the advances in model checking and the chapters of this handbook in terms of two recurrent themes that have driven much of the research agenda in model checking:

1 The algorithmic challenge: *Design model-checking algorithms that scale to real-life problems.* The main practical problem in model checking is the combinatorial explosion of states in the Kripke structure—the "state-explosion problem." Since each state represents the global system status at a given time point, a state is essentially a memory snapshot of the system under investigation, and, thus, the size of the state space is exponential in the size of the memory. Therefore, even for systems of relatively modest size, it is impossible to compute and analyze the entire corresponding Kripke structure directly. In fact, in most situations, the state space is not finite (e.g., unbounded memory, unbounded number of parallel processes), which leads to the modeling challenge. In practice, very large and infinite state-transition systems are approximated by effective abstractions such as finite-state abstractions, or decision procedures are approximated by so-called "semi-algorithms," which are aimed at finding bugs but may fail to prove correctness, or both.

2 The modeling challenge: *Extend the model-checking framework beyond Kripke structures and temporal logic.* Kripke structures are natural representations for various flavors of communicating finite-state machines. To model and specify unbounded iteration and recursion, unbounded concurrency and distribution, process creation and reconfiguration, unbounded data types, real-time and cyber-physical systems, probabilistic computation, security aspects, etc., and to abstract these features effectively, we need to extend the modeling and specification frameworks beyond Kripke structures and classical temporal logics. Some

extensions maintain decidability, often through the construction of finite-state abstractions, which ties the modeling challenge back to the algorithmic challenge. Other extensions sacrifice decidability, but maintain the ability of model checking to find bugs automatically, systematically, and early in the system design process.

As the two challenges are tied together closely, many chapters of this handbook address both. New models without algorithms and without experimental validation are not central to model-checking research.

We believe that the main strengths of model checking are threefold. First, model checking is a systematic, algorithmic methodology which can be computerized and, ideally, fully automated. Thus, model-checking tools have the goal and promise to be used during the design process by hardware and software engineers without the assistance of verification experts. Second, model checking can be applied at different stages of the design process, on abstract models as well as on implemented code. While any one model-checking tool is usually limited to particular modeling and specification languages, the methodology as such is not restricted to any level or formalism and can be applied at different levels, to incomplete systems and partial specifications, to find different kinds of bugs. Third, and perhaps most importantly, model checking deals particularly well with concurrency. The interaction between parallel processes is one of the main sources of complexity and errors in system design. Moreover, concurrency errors are especially subtle, contingent, and therefore difficult to reproduce; they are hard to find by testing the system, and their absence is hard to prove by logical arguments or program analyses because of the extremely large number of possible interactions between parallel processes. Model checking, on the other hand, which is based on the algorithmic exploration of large state spaces, is particularly well suited for finding concurrency bugs and proving their absence.

These strengths distinguish model checking from related approaches for improving system quality:

Testing is the fastest and simplest way to detect errors. It lends itself to easy automation and can often be handled with limited academic background. Testing covers a large spectrum from manual, ad hoc testing of code all the way to automated efforts and model-based testing [37, 48, 50]. Fundamentally dynamic, testing is able to detect compiler errors, hardware errors, and modeling errors that are invisible to static tools; it is integral to any comprehensive approach to safety engineering and software certification [60]. However, *"program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence"* [32], especially for concurrent systems. Model checking, on the other hand, provides a systematic approach to bug detection that can be applied to incomplete systems and, even for concurrent systems, offers in the limit—when no more errors are found—a certificate of correctness. In other words, while testing is only debugging, model checking is systematic debugging aimed at model verification. Chapter 19 will discuss the cross-fertilization of modern model-checking and white-box testing techniques.

Abstract interpretation and other program analyses are similar to model checking in that they employ algorithms to prove program properties automatically [29, 68]. Following the tradition of programming-language semantics rather than logic, static analysis frameworks such as data-flow analysis, abstract interpretation, and rich type systems are built on lattice theory. Historically, their applications have focused mainly on fast analyses for compilers, in particular on overapproximative analyses of numerical data types and assertion violations [6]. Thus, in comparison to model checking, the focus of program analyses is on efficiency rather than expressiveness, and on code rather than models. In recent research on software verification, abstract interpretation and model checking have converged to a large extent. Chapters 15 and 16 will explore the rich relationship between abstract interpretation and model checking in depth.

Higher-order theorem proving is a powerful and proven approach for the verification of complex systems. Semi-manual theorem provers [13, 69, 85] have been used to verify critical systems ranging from floating-point arithmetic [1] to microkernels [51], and even to the proof of deep mathematics such as the Kepler conjecture [40]. In comparison to model checking, the focus of higher-order theorem proving is on expressiveness rather than efficiency: it handles data manipulation precisely and typically aims at full functional correctness. Theorem proving naturally incorporates the manual decomposition of a verification problem, which is obligatory for complex systems. But even for experts, developing proofs in higher-order logic tools is a time-consuming and often non-trivial effort worthy of a research publication. Chapter 20 will explore some connections and combinations of model checking and theorem proving.

While the different verification methods have different historical starting points and different communities, ultimately they simply represent different trade-offs on the efficiency versus expressiveness (or precision) spectrum: greater expressive power tends to take us, at the cost of efficiency, from testing to abstract interpretation to model checking to theorem proving, and, over time, the differences become smaller. Through more than three decades, model checking has acquired and adapted methods from all of these research areas, but also from automata theory, process algebra, graph algorithms, game theory, hardware simulation, stochastic processes, control theory, and many other areas. At the same time, model checking has interacted with target areas for verification, most importantly computer engineering and VLSI design, software engineering and programming languages, embedded and cyber-physical systems, artificial intelligence, and even computational biology. It is thus fair to say that model checking is characterized less by purity of method than by the goal of debugging and analyzing dynamical systems that exist in the real world and can be modeled as state-transition systems.

Model checking has been covered by several monographs [8, 11, 12, 26, 46, 47, 54, 55, 67, 71] and surveys [25, 28, 33]. The early history of model checking is documented in several collections of essays [38, 63]. The rest of this chapter serves as an introduction to the handbook for readers with little familiarity with the material. In Sect. 1.2 we give a minimalist introduction to the classical setting of temporal-logic model checking over Kripke structures. Section 1.3 then revisits

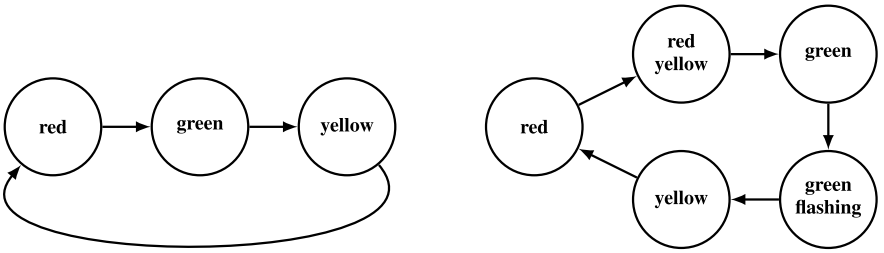


Fig. 2 American and Austrian traffic lights as Kripke structures

some of the fundamental challenges for model checking and puts the individual chapters of the handbook in perspective. In Sect. 1.4 we provide a brief outlook for the field.

1.2 Temporal-Logic Model Checking in a Nutshell

We give a brief introduction to the classical view of temporal-logic model checking.

1.2.1 Kripke Structures

Kripke structures [53] are finite directed graphs whose vertices are labeled with sets of atomic propositions. The vertices and edges of the graph are called "states" and "transitions," respectively. In our context, they are used to represent the possible configurations and configuration changes of a discrete dynamical system. For a simple illustration, consider the two Kripke structures in Fig. 2, which represent the states and state transitions of traffic lights in the USA and Austria, respectively. Formally, a *Kripke structure* over a set A of atomic propositions is a triple $K = \langle S, R, L \rangle$ where S is a finite set of states (the "state space"), $R \subseteq S \times S$ is a set of transitions (the "transition relation"), and the labeling function $L\colon S \to 2^A$ associates each state with a set of atomic propositions. For a state $s \in S$, the set $L(s)$ represents the set of atomic propositions that are true when the system is in state s, and the set $A \setminus L(s)$ contains the propositions that are false in state s. We assume that the transition relation R is total, i.e., that all states have non-zero outdegree.

The dynamic behavior of the system represented by a Kripke structure corresponds to a path through the graph. A path is a finite or infinite sequence $\pi = s_0, s_1, s_2, \ldots$ of states such that $(s_i, s_{i+1}) \in R$ for all $i \geq 0$. The totality of the transition relation ensures that every finite path can be extended to an infinite path. Given an infinite path π, we write $L(\pi) = L(s_0), L(s_1), L(s_2), \ldots$ for the corresponding infinite sequence of sets of atomic propositions. Moreover, we write $\pi^i = s_i, s_{i+1}, s_{i+2}, \ldots$ for the infinite path that results from π by removing the first i states.

Related models that are based on discrete state changes are sometimes called automata, state machines, state diagrams, labeled transition systems, etc. Throughout

this book, such models are used in accordance with the traditions of the respective research areas. In most cases, algorithmic results can be easily transferred between these models. Later handbook chapters will introduce more advanced frameworks—e.g., for modeling recursive (Chap. 17), probabilistic (Chap. 28), and real-time (Chap. 29) behavior—which extend finite state-transition systems in fundamental ways and require more intricate analysis techniques.

1.2.2 The Temporal Logic CTL*

CTL* [34] is a propositional modal logic with path quantifiers, which are interpreted over states, and temporal operators, which are interpreted over paths.

Path quantifiers:

A "for every infinite path from this state"
E "there exists an infinite path from this state"

Temporal operators (for atomic propositions p and q):
$\mathbf{X}p$ "p holds at the next state"
$\mathbf{F}p$ "p holds at some state in the future"
$\mathbf{G}q$ "q holds at all states in the future"
$q\mathbf{U}p$ "p holds at some state in the future, and q holds at all states until p holds"

For instance, $\mathbf{F}p$ holds on path π iff π contains a state with label p, and $\mathbf{A}\varphi$ holds at state s iff φ holds on all infinite paths that start from state s.

Given a set of atomic propositions A, the syntax of CTL* is defined recursively as follows:

- If $p \in A$, then p is a formula of CTL*.
- If φ and ψ are formulas of CTL*, then $\varphi \vee \psi$, $\varphi \wedge \psi$, $\neg\varphi$, $\mathbf{A}\varphi$, $\mathbf{E}\varphi$, $\mathbf{X}\varphi$, $\mathbf{F}\varphi$, $\mathbf{G}\psi$, and $\psi\mathbf{U}\varphi$ are formulas of CTL*.

To reflect the distinction between path quantifiers and temporal operators, we distinguish a syntactic subset of CTL* called state formulas. *State formulas* are boolean combinations of atomic propositions and CTL* formulas whose outermost operator is a path quantifier, i.e., they start with **A** or **E**. As in the example of $\mathbf{A}\varphi$ above, the truth value of a state formula can be asserted over a state in a Kripke structure. For all other formulas of CTL*, we need a path to determine the truth value. The formal semantics of CTL* is based on this syntactic distinction, and presented in Table 1.

An easy exercise on the semantics defined in Table 1 shows that the syntactic restriction of CTL* to one of the path quantifiers and the two temporal operators **X** and **U** yields the full expressive power of CTL*, i.e., all other operators can be defined from these three operators.

Given a Kripke structure K, state s, and state formula f, a model-checking algorithm is a decision procedure for $K, s \models f$. In case of $K, s \not\models f$, many model-checking algorithms provide evidence of the violation of the satisfaction relation,

Table 1 Semantics of CTL*. Here, K is a Kripke structure, π is a path, s is a state, p is an atomic proposition, f and g are state formulas, and φ and ψ are CTL* formulas

$K, s \models p$	iff	$p \in L(s)$
$K, s \models \neg f$	iff	$K, s \not\models f$
$K, s \models f \vee g$	iff	$K, s \models f$ or $K, s \models g$
$K, s \models f \wedge g$	iff	$K, s \models f$ and $K, s \models g$
$K, s \models \mathbf{E}\varphi$	iff	there is an infinite path π starting from s such that $K, \pi \models \varphi$
$K, s \models \mathbf{A}\varphi$	iff	for every infinite path π starting from s we have $K, \pi \models \varphi$
$K, \pi \models f$	iff	$K, s \models f$ for the first state s of π
$K, \pi \models \neg\varphi$	iff	$K, \pi \not\models \varphi$
$K, \pi \models \varphi \vee \psi$	iff	$K, \pi \models \varphi$ or $K, \pi \models \psi$
$K, \pi \models \varphi \wedge \psi$	iff	$K, \pi \models \varphi$ and $K, \pi \models \psi$
$K, \pi \models \mathbf{X}\varphi$	iff	$K, \pi^1 \models \varphi$
$K, \pi \models \mathbf{F}\varphi$	iff	there exists an $i \geq 0$ such that $K, \pi^i \models \varphi$
$K, \pi \models \mathbf{G}\psi$	iff	for all $j \geq 0$ we have $K, \pi^j \models \psi$
$K, \pi \models \psi \mathbf{U} \varphi$	iff	there exists an $i \geq 0$ such that $K, \pi^i \models \varphi$ and for all $0 \leq j < i$ we have $K,\ \pi^j \models \psi$

when possible in the form of a counterexample. If f has the form $\mathbf{A}\varphi$, then a finite or infinite path of K violating φ serves as a counterexample.

Table 2 gives common examples of CTL* formulas. For each formula, the table describes the semantics of the formula and the structure of possible counterexamples, which are illustrated in Fig. 3. Note that a counterexample is a witness for the negated formula. For instance, a counterexample for $\mathbf{AG}p$ is a witness for the satisfaction of $\neg\mathbf{AG}p$, and, thus, for $\mathbf{EF}\neg p$. As line 3 in Table 2 illustrates, a model checker may not be able to give practical counterexamples for formulas with unnegated $\mathbf{E}$ quantifiers. The situation is better for ACTL*, the fragment of CTL* where $\mathbf{E}$ does not occur and negation is restricted to atomic propositions: ACTL* has tree-like counterexamples [27] and plays an important role in abstraction; cf. Chap. 13.

When the specification is satisfied by the structure, the situation is dual: for formulas involving only unnegated $\mathbf{E}$, a model checker can output a witness for the satisfaction of the formula, but for most formulas with unnegated $\mathbf{A}$ this is unrealistic. In the latter situation, vacuity detection can be used as a "sanity check," to check whether the positive verification result may be based on a faulty specification [10]. A classical example of vacuity is antecedent failure, where a specification $\mathbf{A}(\mathbf{G}p \rightarrow \mathbf{F}q)$ holds because $\mathbf{AG}\neg p$ is true.

The example formulas in Table 2 belong to the temporal logics CTL or LTL, which are useful syntactic fragments of CTL*. Intuitively, CTL is a logic based on state formulas, and LTL is a logic avoiding state formulas. While CTL can be model checked particularly efficiently, LTL allows the natural specification of properties of dynamic behaviors, which correspond to paths.

Table 2 Examples of CTL* formulas and respective counterexamples

	Sub-logic	Formula	Intuition	Counterexample
1	CTL, LTL	$\mathbf{AG}p$	p is an invariant	finite path leading to $\neg p$
2	CTL, LTL	$\mathbf{AF}p$	p must eventually hold	infinite path (lasso-shaped) without p
3	CTL, (negated) LTL	$\mathbf{EF}\neg p$	$\neg p$ is reachable	substructure with all reachable states, all containing p
4	CTL, LTL	$\mathbf{AG}(p \vee \mathbf{X}p) = \mathbf{AG}(p \vee \mathbf{AX}p)$	p holds at least every other state	finite path leading to $\neg p$ twice in a row
5	CTL, LTL	$\mathbf{AGF}p = \mathbf{AGAF}p$	p holds infinitely often	infinite path (lasso) on which p occurs only finitely often
6	CTL, LTL	$\mathbf{AG}(p \rightarrow \mathbf{F}q) = \mathbf{AG}(p \rightarrow \mathbf{AF}q)$	every p is eventually followed by q	finite path leading to p, but no q now nor on the infinite path (lasso) afterwards
7	CTL, (boolean combination of) LTL	$(\mathbf{AGF}p) \wedge \mathbf{EF}\neg p$	both 3 and 5 hold	either counterexample for $\mathbf{AGF}p$ or for $\mathbf{EF}\neg p$
8	CTL only	$\mathbf{AGEX}p$	reachability of p in one step is an invariant	finite path leading to a state whose successors all have $\neg p$
9	CTL only	$\mathbf{AG}(p \vee \mathbf{AXAG}q \vee \mathbf{AXAG}\neg q)$	once p does not hold, either q or $\neg q$ become invariant in one step	finite path leading to $\neg p$ from which two finite extensions reach q and $\neg q$
10	LTL only	$\mathbf{AFG}p$	p must eventually become an invariant	infinite path (lasso) on which $\neg p$ occurs infinitely often
11	LTL only	$\mathbf{A}(\mathbf{GF}p \rightarrow \mathbf{GF}q)$	if p holds infinitely often, so does q	infinite path (lasso) on which p occurs infinitely often, but q does not

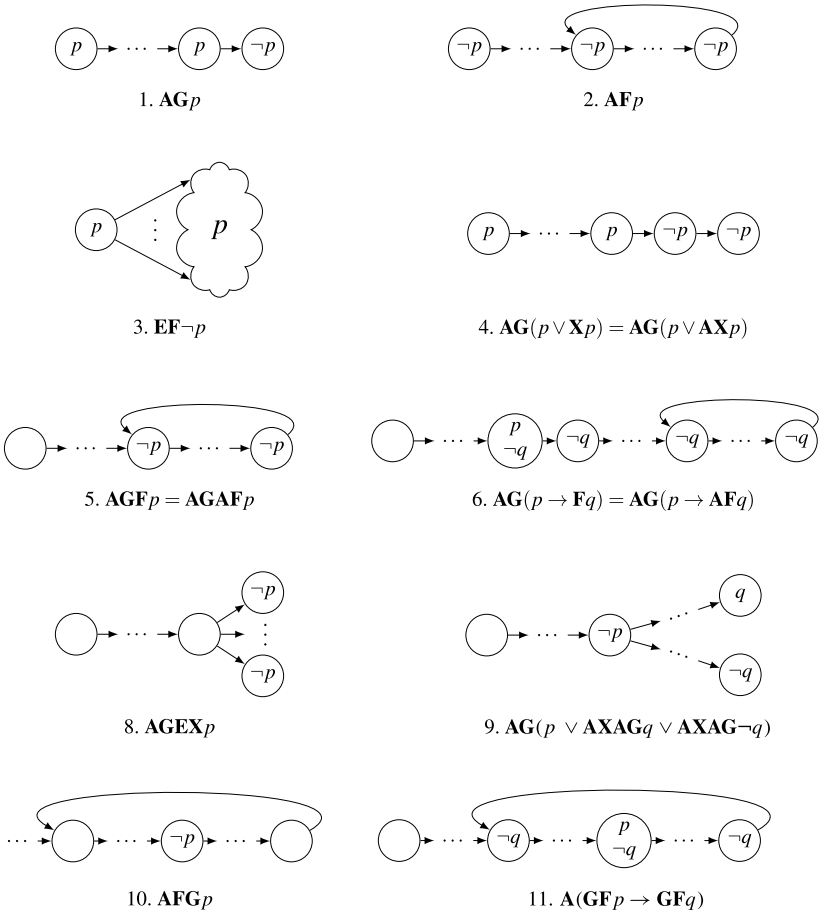


Fig. 3 Graphical illustrations of counterexamples for the formulas from Table 2

1.2.3 The Temporal Logic CTL

CTL (Computation Tree Logic) [22] is the syntactic fragment of CTL* in which every path quantifier is immediately followed by a temporal operator:

- If $p \in A$, then p is a CTL formula.
- If φ and ψ are CTL formulas, then $\varphi \vee \psi$, $\varphi \wedge \psi$, $\neg\varphi$, $\mathbf{AX}\varphi$, $\mathbf{EX}\varphi$, $\mathbf{AF}\varphi$, $\mathbf{EF}\varphi$, $\mathbf{AG}\psi$, $\mathbf{EG}\psi$, $\mathbf{A}\psi\mathbf{U}\varphi$, and $\mathbf{E}\psi\mathbf{U}\varphi$ are CTL formulas.

In other words, CTL can be viewed as a propositional modal logic based on the compound operators **AX**, **EX**, **AF**, **EF**, **AG**, **EG**, **AU**, and **EU**; cf. Fig. 4.

Every CTL formula, and hence also each subformula of a CTL formula, is a state formula. Given a Kripke structure K and a CTL formula φ, we can compute the set $[\![\varphi]\!]_K = \{s : K, s \models \varphi\}$ of states that satisfy φ by a recursive algorithm that

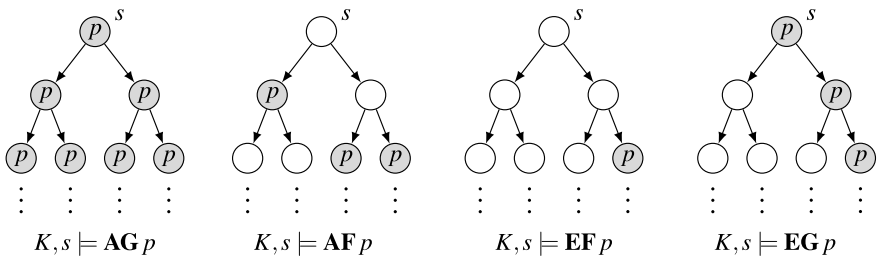


Fig. 4 Examples of CTL operators

first computes $[\![\psi]\!]_K$ for all subformulas ψ of φ. The sets $[\![\psi]\!]_K$ can be seen as a labeling of each state s by subformulas ψ that are true at s; these labelings can be computed in time $O(|K| \times |\varphi|)$. This observation leads directly to the following seminal algorithmic result on CTL model checking.

Theorem 1 ([24]) *There is a CTL model-checking algorithm whose running time depends linearly on the size of the Kripke structure and on the length of the CTL formula* (*if the other parameter is fixed*).

CTL labeling algorithms have a natural formulation as fixed-point computations. For example, the state set $T = [\![\mathbf{EF}\varphi]\!]_K$ can be defined inductively as follows:

- If $K, s \models \varphi$, then $s \in T$.
- If $s \in T$ and there exists a state $s' \in S$ such that $(s', s) \in R$, then $s' \in T$.
- Nothing else is in T.

In other words, T is the smallest set that contains all states labeled by φ and is closed under the **EX** operator. This gives rise to the fixed-point characterization

$$\mathbf{EF}\varphi = (\mu T : \varphi \vee \mathbf{EX}\, T)$$

where μ is the least-fixed-point operator. It is easy to see that all of CTL can be defined as an extension of propositional logic using alternation-free least and greatest fixed points over the temporal operator **EX**. Many advanced CTL model-checking algorithms are based on this fixed-point formulation of CTL; cf. Chap. 8. They typically exploit special data structures and logics for representing and manipulating state sets $[\![\psi]\!]_K$, and use decision procedures for computing **EX** and fixed points (by iteration and subset check). All of CTL* can still be expressed using fixed points over **EX**, but requires the alternation of least and greatest fixed points; the general fixed-point logic called the μ-calculus [52] is discussed in depth in Chap. 26.

Counterexamples for CTL formulas can be very complex. As illustrated in line 3 of Table 2, the simplest counterexample to $K, s \models \mathbf{EF}\, p$ (reachability of a state labeled p) must include all states that are reachable from state s—often, the whole Kripke structure K.

1.2.4 The Temporal Logic LTL

LTL (Linear-time Temporal Logic) [72] is the syntactic fragment of CTL* that contains no path quantifiers except a leading **A**:

- If $p \in A$, then p is an LTL$^-$ formula.
- If φ and ψ are LTL$^-$ formulas, then $\varphi \vee \psi$, $\varphi \wedge \psi$, $\neg\varphi$, $\mathbf{X}\varphi$, $\mathbf{F}\varphi$, $\mathbf{G}\psi$, and $\psi\,\mathbf{U}\varphi$ are LTL$^-$ formulas.
- If ψ is an LTL$^-$ formula, then $\mathbf{A}\psi$ is an LTL formula.

LTL is also called the *linear-time* fragment of the *branching-time* logic CTL*. This is because the LTL$^-$ formulas are interpreted over paths, i.e., over linear sequences of states.

It follows that LTL specifications have simple counterexamples. For a Kripke structure K and LTL$^-$ formula ψ, a counterexample of ψ in K is an infinite path π of K such that $K, \pi \not\models \psi$. Then $K, s \not\models \mathbf{A}\psi$ for the initial state s of π, and, thus, the infinite path π is also a counterexample for the LTL formula $\mathbf{A}\psi$. Moreover, an inductive proof shows that the counterexample π can w.l.o.g. be restricted to have a "lasso" shape $v \cdot w^\omega$, i.e., an initial finite path (prefix) v followed by an infinitely repeated finite path (cycle) w [86].

Certain LTL properties have even simpler counterexamples. The LTL$^-$ formula ψ specifies a *safety property* iff for every Kripke structure K there is a (possibly infinite) set $\Pi_K(\psi)$ of finite paths such that the counterexamples of ψ in K are precisely the infinite extensions of the paths in $\Pi_K(\psi)$. In other words, safety properties always have finite paths as counterexamples: they specify a finite, "bad" series of events that must never happen [56]. The simplest safety properties are invariants, that is, LTL formulas of the form $\mathbf{AG}f$, where f is a boolean combination of atomic propositions. This invariant specifies that the state property f must hold at all reachable states, and a counterexample for $\mathbf{AG}f$ is a finite path whose last state violates f. A more complicated safety property specifies that f must not be followed by g: $\mathbf{AG}(f \rightarrow \mathbf{XG}\neg g)$; in this case, the "bad" series of events that must not happen is an f followed later by a g.

Most LTL properties specify a combination of safety and so-called *liveness properties* [2], and thus may require infinite paths (lassos) as counterexamples. For a detailed discussion of safety versus liveness, see Chaps. 2 and 3. Here we give only two important examples of liveness properties. The LTL$^-$ formula $\mathbf{GF}f$ specifies that, along an infinite path, the state property f holds infinitely often. Obviously, this requirement does not have a finite path as counterexample. The requirement is useful, for instance, for defining a *weakly fair scheduler*. Let p be an atomic proposition that signals that a process is ready to be scheduled, and let q signal that the process is being scheduled. A scheduler is weakly fair if it does not forever neglect scheduling a process that is continuously ready to be scheduled: $\mathbf{GF}(\neg p \vee q)$. By contrast, the LTL$^-$ formula $(\mathbf{GF}p) \rightarrow \mathbf{GF}q$ specifies that, along an infinite path, if the process is ready infinitely often, then it is scheduled infinitely often. This requirement defines a *strongly fair scheduler*, which must not forever neglect scheduling a process that is ready infinitely often (but not necessarily continuously) [64].

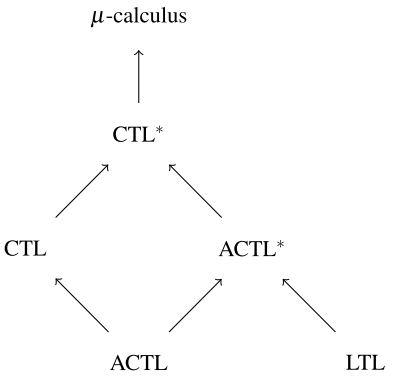


Fig. 5 Basic temporal logics and their relationships. All inclusions are strict. ACTL is the intersection of ACTL* and CTL

The simple structure of counterexamples can be exploited by model-checking algorithms for LTL. The key insight reveals a close relationship between LTL$^-$ formulas and finite automata over infinite words [84]. Using a tableau construction for modal logics, it is possible to translate an LTL$^-$ specification ψ into a Büchi automaton B_ψ over the alphabet 2^A (where A is the set of atomic propositions) such that for all Kripke structures K and infinite paths π, the infinite word $L(\pi)$ is accepted by the automaton B_ψ iff π is a counterexample of ψ in K. The size of the automaton B_ψ is exponential in the length of the formula ψ. From this construction we obtain the following seminal model-checking algorithm for LTL.

Theorem 2 ([61, 65, 86]) *There is an LTL model-checking algorithm whose running time depends linearly on the size of the Kripke structure and exponentially on the length of the LTL formula.*

Technically, the LTL model-checking problem is complete for PSPACE. While this complexity is worse than the complexity of CTL model checking, one should keep in mind that in practice the limiting factor is usually the size of the state space, not the length of the temporal specification.

Chapters 4 and 5 will describe advanced LTL model-checking algorithms, which are based on automata theory and the systematic search for counterexamples. Chapter 10 will present SAT-based model checking, where LTL counterexamples are specified and found by boolean constraint solving.

Figure 5 gives an overview of the temporal logics covered in this section and compares their expressive powers. While this section gave only the briefest introduction to temporal logics, the interested reader should continue with Chap. 2 for more depth.

1.3 A Very Brief Guide Through the Chapters of the Handbook

Over the span of more than three decades, model checking has developed from a niche subject in theoretical computer science into a large family of formalisms,

Table 3 Handbook chapters

Introduction to Model Checking
Temporal Logic and Fair Discrete Systems
Modeling for Verification
Automata Theory and Model Checking
Explicit-State Model Checking
Partial-Order Reduction
Binary Decision Diagrams
BDD-Based Symbolic Model Checking
Propositional SAT Solving
SAT-Based Model Checking
Satisfiability Modulo Theories
Compositional Reasoning
Abstraction and Abstraction Refinement
Interpolation and Model Checking
Predicate Abstraction for Program Verification
Combining Model Checking and Data-Flow Analysis
Model Checking Procedural Programs
Model Checking Concurrent Programs
Combining Model Checking and Testing
Combining Model Checking and Deduction
Model Checking Parameterized Systems
Model Checking Security Protocols
Transfer of Model Checking to Industrial Practice
Functional Specification of Hardware via Temporal Logic
Symbolic Trajectory Evaluation
The mu-Calculus and Model Checking
Graph Games and Reactive Synthesis
Model Checking Probabilistic Systems
Model Checking Real-Time Systems
Verification of Hybrid Systems
Symbolic Model Checking in Non-Boolean Domains
Process Algebra and Model Checking

methods, tools, subcommunities, and application areas. Thus, each chapter of this handbook is a survey in its own right, and reflects the viewpoint, the notation, and the terminology used by the specialists in the respective research area. The handbook chapters are intended as independent introductions to the state-of-the-art research literature on specific topics rather than as chapters of a monograph. Having said this, the order of the chapters was chosen so that the handbook can, in principle, be read cover to cover as if it were a monograph. We have tried to order the chapters

so that the prerequisite background of each chapter occurs in earlier chapters, thus avoiding dependencies on later chapters as much as possible. We also encourage the browsing of neighboring chapters, which often discuss related topics.

The first few chapters of the handbook cover the basics of model checking. Chapter 2 introduces the temporal logics that are commonly used as specification languages in model checking and were outlined in Sect. 1.2. Chapter 3 shows how Kripke structures can be used to model a wide variety of different software and hardware systems. Chapters 4–5 constitute a mini-course in explicit-state LTL model checking or, more generally, the automata-theoretic approach to model checking. As was argued in Sect. 1.2.4, the simple structure of LTL counterexamples motivates model-checking algorithms that search for counterexamples. Explicit-state model checking can be understood as a graph-theoretic search procedure for counterexamples that uses a finite automaton which monitors the truth of the specification.

In the rest of this section, we give a brief preview of the remaining chapters through the lens of the classification "Algorithmic Challenge" versus "Modeling Challenge," which was put forward in Sect. 1.1. The two categories are not exclusive, as many chapters address both challenges. Due to the breadth of the covered material, we refrain in this section from references to the literature, leaving all citations to the individual chapters.

1.3.1 The Algorithmic Challenge

A significant part of model-checking research has focused on algorithmic methods to deal with state explosion. These methods avoid the explicit construction of the complete Kripke structure. We can roughly classify these methods into three groups:

Structural methods for model checking exploit the structure of the syntactic expression (the "code") that defines the system. Large hardware and software systems are described modularly using, for example, subroutines (procedure and method calls) for sequential structuring, and interacting parallel hardware components and software processes (threads, actors) for concurrent structuring. While the state space may be finite, it can be extremely large, and "flattening" the system description—i.e., constructing and exploring the Kripke structure which represents the entire state space—would sacrifice any advantages, such as symmetries, that can be obtained from studying the definition of the system. Moreover, in many cases, such as recursive procedure calls or the concurrent composition of a parametric number of processes, the number of states is unbounded and the full Kripke structure cannot be computed. Techniques such as symmetry reduction, on-the-fly state-space exploration, partial-order reduction, assume-guarantee reasoning, and parametric verification avoid mindless flattening and, in one way or another, make use of the system structure for better performance.

The limiting factor for exhaustive state-space exploration (Chap. 5) is usually memory space, even if new system states are generated during the search from

the system definition only as needed ("on the fly"). Chapter 6 presents a search optimization technique for concurrent software systems, whose complexity stems from the large number of possible interleavings of concurrent processes. The so-called partial-order reduction exploits the fact that the ordering of independent events from different processes is not important for the result of a computation. Compositional reasoning (Chap. 12) exploits the modular definition of complex systems through a divide-and-conquer approach that puts together a proof of overall system correctness from (generally simpler) correctness proofs about system parts. Often the proofs about system parts cannot be entirely independent, though, and a proof about one component of the system may make assumptions about the behavior of other components; this is called "assume-guarantee reasoning." Chapter 17 discusses the verification of software with procedure calls, which draws on the theory of push-down systems. Chapter 18 discusses the verification of software with multiple concurrent processes that synchronize on shared memory (e.g., through locks) or through message passing. Chapter 21 discusses the verification of systems with an unknown number of identical concurrent processes ("parametric verification").

Symbolic methods represent state sets and the transition relation of a Kripke structure by an expression in a symbolic logic, rather than by an explicit enumeration of states or transitions. Symbolic encodings—be it through binary decision diagrams, propositional formulas, or quantifier-free first-order constraints—can result in a dramatic compression of the data structures for representing state sets and, if the necessary operations can be performed efficiently, in order-of-magnitude improvements in the practical performance of verification tools. Chapters 7 and 8 are devoted to model checking with binary decision diagrams (BDDs). As a data structure for boolean functions, BDDs have the advantage over boolean formulas and circuits that satisfiability and equivalence can be checked in constant time. When used in the fixed-point algorithms for CTL that were discussed in Sect. 1.2.3, BDD-based encodings of state sets often achieve in practice an exponential reduction in the size of the data structures. The dramatic performance improvements obtained by BDD-based model checking in the 1990s were essential for the success of model checking in the hardware industry.

While originally the term "symbolic model checking" was used synonymously with BDD-based model checking, more recently other symbolic encodings of state sets and paths—both in the boolean and more general cases—have proved useful in different circumstances. Chapters 9 and 10 present SAT-based model checking, a symbolic model-checking method for LTL. As was discussed in Sect. 1.2.4, LTL has lasso-shaped counterexamples, and boolean formulas can be used to specify the constraints for the existence of a counterexample in terms of propositional logic. By solving these constraints, a boolean satisfiability solver can compute the constituting parts of the counterexample, or disprove the existence of a counterexample. In this way, SAT-based model checkers can profit from recent improvements in SAT solving. While in its original formulation, SAT-based model checking was incomplete, because it could find only counterexample paths of bounded length ("bounded model checking"), Chap. 10 discusses methods to achieve completeness. Today, SAT-based model checking is a standard tool

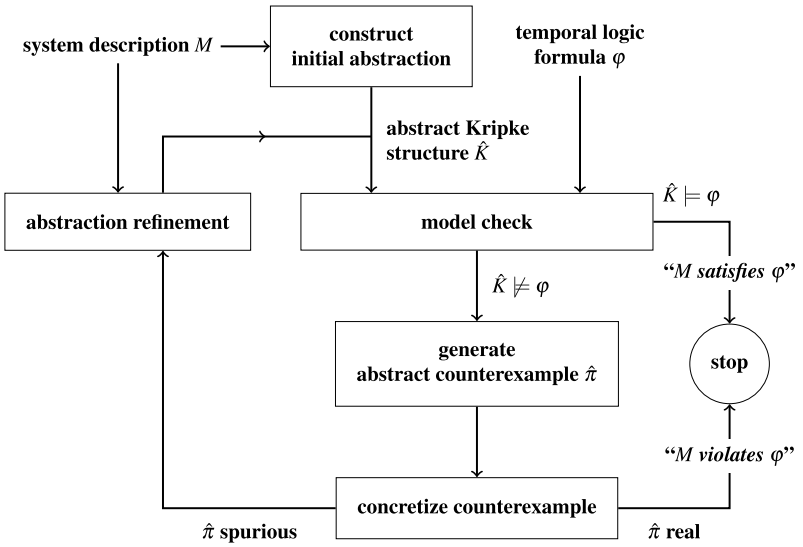


Fig. 6 Counterexample-guided abstraction refinement

in the hardware industry and is also used in bit-precise software model checking. For expressing states over non-boolean domains, boolean logic can be replaced by decidable theories of quantifier-free first-order logic, a.k.a. SAT modulo theories (SMT). SMT solvers (Chap. 11) are powerful decision procedures which have evolved rapidly and underlie many modern software model checkers. General fixed-point algorithms for model checking in boolean and non-boolean domains are discussed in Chaps. 26 and 31.

Abstraction (Chapters 13–15) is a more aggressive approach to state explosion. Abstraction reduces a Kripke structure K to a smaller homomorphic image $\hat{K}$—the abstract model—which preserves certain properties of the original structure and can be analyzed more efficiently. In other words, the abstract model is a principled overapproximation of the system. For example, the existence of a simulation relation between the original and abstract structures guarantees that tree-shaped counterexamples on the original structure are not lost in the abstraction, although there may be "spurious counterexamples" which occur only in the abstraction. Formally, if $\hat{K}$ simulates K, then for all ACTL* specifications φ, if $\hat{K} \models \varphi$ then $K \models \varphi$, but not vice versa. Other relationships between the original and abstract structures preserve different specification logics; see Chap. 13.

A key ingredient of many modern model checkers is counterexample-guided abstraction refinement—an algorithmic method for verifying a system by constructing iterated abstractions of increasing precision. To this end, spurious counterexamples are analyzed and eliminated by adding to the abstract model previously neglected details from the system description, in order to improve the abstraction until either a real counterexample—i.e., a bug—is found, or no more spurious counterexamples occur and the system is verified; see Fig. 6. Chapter 14 discusses

interpolation, a paradigmatic logical method for localizing within paths assertions made in proofs, which can be used to identify and eliminate spurious counterexamples. Chapter 15 presents a modern introduction to software verification based on predicate abstraction, where states of the abstract model are described using constraints ("predicates") on the program counter and program variables.

As was discussed in Sect. 1.1, model checking is closely related to other algorithmic and semi-algorithmic approaches for system correctness, such as testing, program analysis, and theorem proving. All of these methods have cross-fertilized each other and are often combined; this is explained in the three chapters that discuss combinations of model checking and data-flow analysis (Chap. 16), testing (Chap. 19), and deduction (Chap. 20).

1.3.2 The Modeling Challenge

While Chap. 3 illustrates a broad range of systems that can be modeled using Kripke structures, for some critical applications the basic state-transition system model must be augmented with additional features. The chapters of this handbook discuss several such extensions. Besides non-boolean data (Chap. 16), recursion (Chap. 17), and an unbounded number of parallel processes (Chap. 21) for modeling software, there are four paradigmatic extensions of finite state-transition systems that are essential for modeling certain important classes of systems.

Security protocols provide a perfect application domain for formal methods: they are often small but difficult to get right, and their correctness is critical. However, any method for security protocols must handle non-finite-state concepts such as nonces and keys, encryption and decryption, and unknown attackers; see Chap. 22.

Graph games are an extension of state-transition systems with multiple actors. In each state, one or more of the actors choose actions that, taken together, determine the next state of the system. Graph games are needed to model systems with multiple components, processes, actors, or agents that have different, sometimes conflicting objectives. Even if the system is monolithic, its environment must sometimes be considered independently, as an adversary in a two-player game. The theory of graph games, which is presented in Chap. 27, studies the strategies that the players can employ to reach their objectives, which also provides the mathematical foundation for synthesizing systems that realize a desired input/output behavior ("reactive synthesis").

Probabilistic systems are state-transition systems, such as discrete-time Markov chains or Markov decision processes, where from certain states the next state is chosen according to a probability distribution. Probabilistic systems can model uncertainty. The model checking of probabilistic systems is discussed in Chap. 28.

Real-time and hybrid systems are extensions of discrete state-transition systems with continuous components. In real-time systems, the continuous components are clocks, which measure and constrain the times at which state transitions may happen; the extension of finite automata with such clocks and clock constraints is called "timed automata." In hybrid systems, finite automata can be extended with more general continuous variables, representing, for example, the location or temperature of a physical system. While timed automata have become a standard model for continuous-time state-transition systems, hybrid automata are needed for modeling physical systems that are controlled by hardware and software ("cyber-physical systems"). Certain properties of timed and hybrid automata remain decidable; in other cases we have semi-algorithms for model checking, which compute the answer to a verification question but in some cases may fail to terminate. The model checking of real-time systems is discussed in Chap. 29; of hybrid systems, in Chap. 30.

Combinations of these extensions are needed for certain modeling tasks, such as stochastic games or continuous-time Markov systems.

Three of the handbook chapters discuss the special needs of hardware verification—the application area in which model checking, as a close relative of equivalence checking, made its first inroads into industry. Chapter 23 provides the history and special challenges of formal verification in hardware, and thus explains a successful example of technology transfer from academia to industry. Chapter 24 discusses the special requirements on specification languages for hardware verification. Chapter 25 presents symbolic trajectory evaluation—a symbolic method specific to hardware validation which is based on abstraction in three-valued logic.

The final chapter of the handbook (Chap. 32) presents process algebra, a formal expression framework for modeling interacting concurrent processes. While originally developed for operational, algebraic, and axiomatic reasoning about concurrent processes, and later extended to cope with constructs for process creation and mobility, model checking quickly became the analysis method of choice for finite-state fragments of process algebra. This pattern has been typical for the practical use of model checking: it can provide a powerful tool for rapid prototyping and debugging, even when full-fledged correctness proofs may require more expressive formalisms.

1.4 The Future of Model Checking

The ubiquity and complexity of hardware- and software-based systems continue to grow. We see ever-increasing levels of concurrency, from multi-core processors to data centers, sensor networks, and the cloud. In addition, hardware- and software-based systems are increasingly deployed in safety-critical situations, to control and connect physical systems from cardiac pacemakers to aircraft. With this dramatic growth in systems complexity and criticality grows also the need for effective verification techniques. In other words, the opportunities for model checking abound.

We expect the corresponding research efforts to continue to make progress with regard to both the scalability and the modeling challenges, to bring ever larger and more varied verification problems within the reach of model-checking technology. Model checking has already been integrated into the design process for hardware; see Chap. 23. In the near future we expect model checking to make similar inroads into the practice of software development, especially in error-prone control-centric—as opposed to data-centric—software domains, including systems software (kernels, schedulers, device drivers, memory and communication protocols, etc.), distributed algorithms and concurrent data structures, and the rapidly growing areas of software-defined networking [17, 70] and cyber-physical systems [3, 57, 81].

Second, while in the past new technology was driven largely by gains in performance (Moore's law) and functionality (new features), we suggest that in the future the correctness, reliability, security, and overall robustness of systems will play an ever-increasing role, also as a differentiating factor for software and hardware products. We see this trend already in modern software systems research, where performance used to be the dominant criterion: correctness has moved recently to center stage in compilers [59], operating systems [51, 87], and distributed systems research [31]. This trend will further increase the impact of formal verification techniques such as model checking.

We suggest that there is a third, even more basic underlying reason for the growing importance of model checking, which stems from the emergence of state-transition systems as a universal model for the design and study of computer-controlled dynamical systems. The state-transition system is the discrete analogue to the continuous dynamical system. With the rise of digital technology, models based on state-transition systems (or "discrete-event systems" [18]) have become ubiquitous in systems engineering. They are equally well suited for formalizing other dynamic processes designed by humans, including all kinds of charts, diagrams, and rules for workflows, interactions, and adaptive structures that occur in organizations [9, 30, 41, 49, 83], and for defining discrete abstractions of continuous processes that occur in the physical world [5]. Model checking—as the central paradigm for analyzing state-transition systems—is therefore only at the beginning of its application to a wide range of different domains, from engineering to science, business, law, etc.

In the following, we highlight, as examples, two of the many currently active directions of model-checking research, as well as two of the many potential new application areas for model checking.

In a first trend, much current research is devoted to moving beyond verification to *synthesis*. Verification is the task of checking whether a given system satisfies a given specification; synthesis is the task of constructing a system that satisfies a given specification. While fully automatic functional synthesis is practical only in constrained situations—such as compilation from a high-level language, or circuit synthesis from given building blocks—the general synthesis task has seen much recent progress in settings where it is required to refine or complete a given, partial system description in order to satisfy certain properties. Often the synthesized properties are non-functional; for example, a sequential program can be automatically

equipped with synchronization constructs such as locks or atomic regions in order to make the program safe for concurrent execution without changing its sequential semantics [19]. More generally, the goal of "computer-aided programming" [80] is to relieve the programmer from tedious but error-prone implementation details so that they can focus on the functionality of the design, while automating the fulfillment of other requirements on the code such as security and fault tolerance. Reactive (or sequential) synthesis refers to the synthesis of state-transition systems that satisfy a given temporal specification, which constrains the input/output behavior of the desired system; this problem has parallel histories in mathematical logic, control theory, game theory, and reactive programming [73, 76, 77], and is discussed in detail in Chap. 27. More recently, much effort has been devoted to template- and syntax-guided approaches to synthesis [4], and to the use of inductive and learning techniques in synthesis [39, 42, 78].

A second important trend that receives much current attention generalizes the classical boolean setting of model checking towards a quantitative setting of *model measuring*. Model checking answers the boolean question of whether a system does or does not satisfy a specification; model measuring quantifies the quality of a system, e.g., by computing a distance between the system and the specification [43, 44]. Quantitative measures can be used to distinguish different systems that satisfy the same functional specification: they may measure the performance of the system, its resource consumption, its cost, or other non-functional properties such as different notions of robustness. Quantitative measures can also be used to distinguish different systems that do not satisfy a given specification; for instance, a system that violates the specification only in rare circumstances, or infrequently, or far in the future is usually preferred to a system that violates the specification in all cases, all the time. While absolute correctness is required in certain situations, "best effort" may be acceptable in others. Since logics with a boolean semantics cannot distinguish between different "degrees" of violation of the satisfaction relation, the classical model-checking framework needs to be extended in order to capture quantitative nuances. Several such extensions have been proposed, including quantitative temporal logics [15]. Statistical model checking [58] replaces absolute guarantees with probabilistic guarantees. Quantitative distance measures between systems [20] generalize the boolean paradigms of system equivalences and refinement preorders, which have been central to many theoretical and practical developments in formal methods: combinational and sequential equivalence checking for hardware, stepwise refinement for systems development, and abstraction refinement for systems analysis. For example, quantitative abstraction refinement can be used to approximate the worst-case execution time of a system to any desired degree of precision [21].

A recent, perhaps unexpected application scenario for model checking is the modeling and analysis of *biological phenomena* using state-transition systems [35, 74]. On a molecular level, biochemical reactions can be modeled faithfully as continuous-time stochastic processes. Yet on a higher level, some mechanistic and organizing principles of biology may be better captured and explained using interacting discrete events, i.e., state-transition abstractions. In fact, it is tempting to

postulate a tower of biological abstraction layers, from molecular pathways to cells to organs and organisms, in analogy to the tower of abstraction layers that tame the design and understanding of complex digital systems: the net-list of boolean gates, the register-transfer level, the instruction set architecture, the high-level programming language. While much work in this direction remains speculative, it is time for scientists to become familiar with the power of discrete state-transition models and the model-checking paradigm, in the same way in which differential-equation models and numerical simulation have become standard tools of the sciences.

An equally speculative and perhaps even more important new application domain for model checking is the analysis of *learning-based software* and other systems built on modern artificial intelligence. While traditional, logic-based AI had a direct link to formal verification based on the common foundation of mathematical logic, many of the successes of modern AI seem to escape rational explanation and quantitative analysis. While there are quantitative methods based on probabilistic models, mathematical statistics, and optimization for attempting to explain and quantify machine-learning systems, the guarantee of properties of such systems will require a more formal approach. As many learning-based systems—such as those used for computer vision in self-driving cars—are safety-critical, even limited formal guarantees would have immense value. A demand for verifiability may guide the design of such systems, for example, by incorporating monitoring mechanisms that ensure that the system does not leave a safe envelope of operation [14]. Dually, learning-based techniques have already started to enter model-checking technology [16]. Once again, this shows that model checking is a vibrant, expanding field of research with no lack of challenges and impact for many years to come.

Acknowledgements The authors thank Guy Avni, Roderick Bloem, Nicolas Braud-Santoni, Andrey Kupriyanov, and Shaz Qadeer for suggestions to improve the text, and Andrey for help with the figures. This work was supported in part by the European Research Council (ERC) and the Austrian Science Fund (FWF).

References

1. Akbarpour, B., Abdel-Hamid, A.T., Tahar, S., Harrison, J.: Verifying a synthesized implementation of IEEE-754 floating-point exponential function using HOL. Comput. J. **53**(4), 465–488 (2010)
2. Alpern, B., Schneider, F.B.: Defining liveness. Inf. Process. Lett. **21**(4), 181–185 (1985)
3. Alur, R.: Principles of Cyber-Physical Systems. MIT Press, Cambridge (2015)
4. Alur, R., Bodík, R., Juniwal, G., Martin, M.M.K., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: Jobstmann, B., Ray, S. (eds.) Proceedings, Formal Methods in Computer-Aided Design, FMCAD, Portland, OR, USA, October 20–23, 2013, pp. 1–8. IEEE, Piscataway (2013)
5. Alur, R., Henzinger, T.A., Lafferriere, G., Pappas, G.J.: Discrete abstractions of hybrid systems. Proc. IEEE **88**(7), 971–984 (2000)
6. Appel, A.W.: Modern Compiler Implementation in C. Cambridge University Press, Cambridge (1998)
7. Apt, K.R., de Boer, F.S., Olderog, E.-R.: Verification of Sequential and Concurrent Programs. Texts in Computer Science. Springer, Heidelberg (2009)

8. Baier, C., Katoen, J-P.: Principles of Model Checking. MIT Press, Cambridge (2008)
9. Baresi, L., Di Nitto, E.: Test and Analysis of Web Services. Springer, Heidelberg (2007)
10. Beer, I., Ben-David, S., Eisner, C., Rodeh, Y.: Efficient detection of vacuity in ACTL formulas. In: Grumberg, O. (ed.) Proceedings, Computer Aided Verification, CAV, Haifa, Israel, June 22–25, 1997. LNCS, vol. 1254, pp. 279–290. Springer, Heidelberg (1997)
11. Ben-Ari, M.: Principles of the SPIN Model Checker. Springer, Heidelberg (2008)
12. Bérard, B., Bidoit, M., Finkel, A., Laroussinie, F., Petit, A., Petrucci, L., Schnoebelen, P.: Systems and Software Verification: Model-Checking Techniques and Tools. Springer, Heidelberg (2001)
13. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Springer, Heidelberg (2004)
14. Bloem, R., Könighofer, B., Könighofer, R., Wang, C.: Shield synthesis: runtime enforcement for reactive systems. In: Baier, C., Tinelli, C. (eds.) Proceedings, Tools and Algorithms for the Construction and Analysis of Systems, TACAS, London, UK, April 11–18, 2015. LNCS, vol. 9035, pp. S33–S48. Springer, Heidelberg (2015)
15. Boker, U., Chatterjee, K., Henzinger, T.A., Kupferman, O.: Temporal specifications with accumulative values. ACM Trans. Comput. Log. **15**(4), 27:1–27:25 (2014)
16. Bortolussi, L., Milios, D., Sanguinetti, G.: Machine-learning methods in statistical model checking and system design. In: Proceedings, Runtime Verification, RV, Vienna, Austria, September 22–25, 2015. LNCS, vol. 9333, pp. 323–341. Springer, Heidelberg (2015)
17. Canini, M., Venzano, D., Perešíni, P., Kostić, D., Rexford, J.: A NICE way to test openflow applications. In: Gribble, S.D., Katabi, D. (eds.) Proceedings, Networked Systems Design and Implementation, NSDI, San Jose, CA, USA, April 25–27, 2012, pp. 127–140. USENIX Association, Berkeley (2012)
18. Cassandras, C.G., Lafortune, S.: Introduction to Discrete-Event Systems, 2nd edn. Springer, Heidelberg (2008)
19. Cerný, P., Clarke, E.M., Henzinger, T.A., Radhakrishna, A., Ryzhyk, L., Samanta, R., Tarrach, T.: From non-preemptive to preemptive scheduling using synchronization synthesis. In: Kroening, D., Pasareanu, C.S. (eds.) Proceedings, Computer Aided Verification, CAV, Part II, San Francisco, CA, USA, July 18–24, 2015. LNCS, vol. 9207, pp. 180–197. Springer, Heidelberg (2015)
20. Cerný, P., Henzinger, T.A., Radhakrishna, A.: Simulation distances. Theor. Comput. Sci. **413**(1), 21–35 (2012)
21. Cerný, P., Henzinger, T.A., Radhakrishna, A.: Quantitative abstraction refinement. In: Giacobazzi, R., Cousot, R. (eds.) Proceedings, Principles of Programming Languages, POPL, Rome, Italy, January 23–25, 2013, pp. 115–128. ACM, New York (2013)
22. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching-time temporal logic. In: Kozen, D. (ed.) Proceedings, Logics of Programs, Yorktown Heights, NY, USA, May 1981. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1981)
23. Clarke, E.M., Emerson, E.A., Sifakis, J.: Model checking: algorithmic verification and debugging. Commun. ACM **52**(11), 74–84 (2009)
24. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite state concurrent systems using temporal logic specifications: a practical approach. In: Wright, J.R., Landweber, L., Demers, A.J., Teitelbaum, T. (eds.) Proceedings, Principles of Programming Languages, POPL, Austin, TX, USA, January 1983, pp. 117–126. ACM, New York (1983)
25. Clarke, E.M., Fehnker, A., Jha, S.K., Veith, H.: Temporal-logic model checking. In: Hristu-Varsakelis, D., Levine, W.S. (eds.) Handbook of Networked and Embedded Control Systems, pp. 539–558. Birkhäuser, Basel (2005)
26. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
27. Clarke, E.M., Jha, S., Lu, Y., Veith, H.: Tree-like counterexamples in model checking. In: Proceedings, Logic in Computer Science, LICS, Copenhagen, Denmark, July 22–25 July 2002, pp. 19–29. IEEE, Piscataway (2002)

28. Clarke, E.M., Schlingloff, B.-H.: Model checking. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, pp. 1635–1790. Elsevier/MIT Press, Amsterdam/Cambridge (2001)
29. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Graham, R.M., Harrison, M.A., Sethi, R. (eds.) Principles of Programming Languages, POPL, Los Angeles, CA, USA, January 1977, pp. 238–252. ACM, New York (1977)
30. Damm, W., Harel, D.: LSCs: breathing life into message sequence charts. Form. Methods Syst. Des. **19**(1), 45–80 (2001)
31. Desai, A., Gupta, V., Jackson, E.K., Qadeer, S., Rajamani, S.K., Zufferey, D.: P: safe asynchronous event-driven programming. In: Boehm, H-J., Flanagan, C. (eds.) Proceedings, Programming Language Design and Implementation, PLDI, Seattle, WA, USA, June 16–19, 2013, pp. 321–332. ACM, New York (2013)
32. Dijkstra, E.W.: The humble programmer. Commun. ACM **15**(10), 859–866 (1972)
33. Emerson, E.A.: Temporal and modal logic. In: Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics, pp. 995–1072. MIT Press, Cambridge (1990)
34. Emerson, E.A., Halpern, J.Y.: "Sometimes" and "not never" revisited: on branching versus linear-time temporal logic. J. ACM **33**(1), 151–178 (1986)
35. Fisher, J., Harel, D., Henzinger, T.A.: Biology as reactivity. Commun. ACM **54**(10), 72–82 (2011)
36. Floyd, R.W.: Assigning meaning to programs. In: Schwartz, J.T. (ed.) Proceedings, Mathematical Aspects of Computer Science: American Mathematical Society Symposia, Providence, RI, USA, vol. 19, pp. 19–31. AMS, Providence (1967)
37. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: Sarkar, V., Hall, M.W. (eds.) Proceedings, Programming Language Design and Implementation, PLDI, Chicago, IL, USA, June 12–15, 2005, pp. 213–223. ACM, New York (2005)
38. Grumberg, O., Veith, H.: 25 Years of Model Checking: History, Achievements, Perspectives. LNCS, vol. 5000. Springer, Heidelberg (2008)
39. Gulwani, S., Hernández-Orallo, J., Kitzelmann, E., Muggleton, S.H., Schmid, U., Zorn, B.G.: Inductive Programming meets the real world. Commun. ACM **58**(11), 90–99 (2015)
40. Hales, T.C., Adams, M., Bauer, G., Dang, D.T., Harrison, J., Hoang, T.L., Kaliszyk, C., Magron, V., McLaughlin, S., Nguyen, T.T., Nguyen, T.Q., Nipkow, T., Obua, S., Pleso, J., Rute, J., Solovyev, A., Ta, A.H.T., Tran, T.N., Trieu, D.T., Urban, J., Vu, K.K., Zumkeller, R.: A formal proof of the Kepler conjecture. CoRR (2015). arXiv:1501.02155 [abs]
41. Harel, D.: Statecharts: a visual formalism for complex systems. Sci. Comput. Program. **8**(3), 231–274 (1987)
42. Harel, D., Marelly, R.: Come, Let's Play. Scenario-Based Programming Using LSCs and the Play-Engine. Springer, Heidelberg (2003)
43. Henzinger, T.A.: Quantitative reactive modeling and verification. Comput. Sci. Res. Dev. **28**(4), 331–344 (2013)
44. Henzinger, T.A., Otop, J.: From model checking to model measuring. In: D'Argenio, P.R., Melgratti, H.C. (eds.) Proceedings, Concurrency Theory, CONCUR, Buenos Aires, Argentina, August 27–30, 2013. LNCS, vol. 8052, pp. 273–287. Springer, Heidelberg (2013)
45. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
46. Holzmann, G.J.: Design and Validation of Computer Protocols. Prentice Hall, New York (1995)
47. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley, Reading (2004)
48. Jacky, J., Veanes, M., Campbell, C., Schulte, W.: Model-Based Software Testing and Analysis with C#. Cambridge University Press, Cambridge (2007)
49. Karamanolis, C.T., Giannakopoulou, D., Magee, J., Wheater, S.M.: Model checking of workflow schemas. In: Proceedings, Enterprise Distributed Object Computing, EDOC, Makuhari, Japan, September 25–28, 2000, pp. 170–181. IEEE, Piscataway (2000)

50. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976)
51. Klein, G., Andronick, J., Elphinstone, K., Murray, T.C., Sewell, T., Kolanski, R., Heiser, G.: Comprehensive formal verification of an OS micro-kernel. ACM Trans. Comput. Syst. **32**(1), 2 (2014)
52. Kozen, D.: Results on the propositional μ-calculus. Theor. Comput. Sci. **27**(3), 333–354 (1983)
53. Kripke, S.: A completeness theorem in modal logic. J. Symb. Log. **24**(1), 1–14 (1959)
54. Kropf, T.: Introduction to Formal Hardware Verification. Springer, Heidelberg (1999)
55. Kurshan, R.P.: Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach. Princeton University Press, Princeton (1994)
56. Lamport, L.: Proving the correctness of multiprocess programs. IEEE Trans. Softw. Eng. **2**, 125–143 (1977)
57. Lee, E.A., Seshia, S.: Introduction to Embedded Systems, A Cyber-physical Systems Approach, 2nd edn. MIT Press, Cambridge (2015)
58. Legay, A., Delahaye, B., Bensalem, S.: Statistical model checking: an overview. In: Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G.J., Rosu, G., Sokolsky, O., Tillmann, N. (eds.) Proceedings, Runtime Verification, RV, St. Julian's, Malta, November 1–4, 2010. LNCS, vol. 6418, pp. 122–135. Springer, Heidelberg (2010)
59. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7), 107–115 (2009)
60. Leveson, N.G.: Safeware: System Safety and Computers. Addison-Wesley, Reading (1995)
61. Lichtenstein, O., Pnueli, A.: Checking that finite-state concurrent programs satisfy their linear specification. In: Van Deusen, M.S., Galil, Z., Reid, B.K. (eds.) Principles of Programming Languages, POPL, New Orleans, LA, USA, January 1985, pp. 97–107. ACM, New York (1985)
62. Manna, Z.: Introduction to Mathematical Theory of Computation. McGraw-Hill, New York (1974)
63. Manna, Z., Peled, D. (eds.): Time for Verification, Essays in Memory of Amir Pnueli. LNCS, vol. 6200. Springer, Heidelberg (2010)
64. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems: Specification. Springer, Heidelberg (1992)
65. Manna, Z., Wolper, P.: Synthesis of communicating processes from temporal-logic specifications. In: Kozen, D. (ed.) Proceedings, Logics of Programs, Yorktown Heights, NY, USA, May 1981. LNCS, vol. 131, pp. 253–281. Springer, Heidelberg (1981)
66. McCarthy, J.: A basis for a mathematical theory of computation. In: Braffort, P., Hirschberg, D. (eds.) Computer Programming and Formal Systems. Studies in Logic and the Foundations of Mathematics, vol. 35, pp. 33–70. Elsevier, Amsterdam (1963)
67. McMillan, K.L.: Symbolic Model Checking. Springer, Heidelberg (1993)
68. Nielson, F., Riis Nielson, H., Hankin, C.: Principles of Program Analysis. Springer, Heidelberg (1999)
69. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. Springer, Heidelberg (2002)
70. Panda, A., Argyraki, K.J., Sagiv, M., Schapira, M., Shenker, S.: New directions for network verification. In: Ball, T., Bodík, R., Krishnamurthi, S., Lerner, B.S., Morrisett, G. (eds.) Proceedings, Summit on Advances in Programming Languages, SNAPL, Asilomar, CA, USA, May 3–6, 2015. LIPIcs, vol. 32, pp. 209–220. Schloss Dagstuhl, Wadern (2015)
71. Peled, D.A.: Software Reliability Methods. Springer, Heidelberg (2001)
72. Pnueli, A.: The temporal logic of programs. In: Proceedings, Foundations of Computer Science, FOCS, Providence, RI, USA, October 31–November 1, 1977, pp. 46–57. IEEE, Piscataway (1977)
73. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Proceedings, Principles of Programming Languages, POPL, Austin, TX, USA, January 11–13, 1989, pp. 179–190. ACM, New York (1989)
74. Priami, C., Morine, M.J.: Analysis of Biological Systems. Imperial College Press, London (2015)

75. Queille, J.-P., Sifakis, J.: Specification and verification of concurrent systems in CESAR. In: Dezani-Ciancaglini, M., Montanari, U. (eds.) Proceedings, International Symposium on Programming, Torino, Italy, April 6–8, 1982. LNCS, vol. 137, pp. 337–351. Springer, Heidelberg (1982)
76. Rabin, M.O.: Automata on Infinite Objects and Church's Problem. AMS, Providence (1972)
77. Ramadge, P.J.G., Wonham, W.M.: The control of discrete event systems. Proc. IEEE **77**(1), 81–98 (1989)
78. Raychev, V., Bielik, P., Vechev, M.T., Krause, A.: Learning programs from noisy data. In: Bodík, R., Majumdar, R. (eds.) Proceedings, Principles of Programming Languages, POPL, St. Petersburg, FL, USA, January 20–22, 2016, pp. 761–774. ACM, New York (2016)
79. Rice, H.G.: Classes of recursively enumerable sets and their decision problems. Trans. Am. Math. Soc. **74**, 358–366 (1953)
80. Solar-Lezama, A.: Program sketching. Softw. Tools Technol. Transf. **15**(5–6), 475–495 (2013)
81. Tabuada, P.: Verification and Control of Hybrid Systems. Springer, Heidelberg (2009)
82. Turing, A.: On computable numbers, with an application to the Entscheidungsproblem. Proc. Lond. Math. Soc. **42**, 230–265 (1937)
83. van der Aalst, W.M.P.: Process Mining—Data Science in Action, 2nd edn. Springer, Heidelberg (2016)
84. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification. In: Proceedings, Logic in Computer Science, LICS, Cambridge, MA, USA, June 16–18, 1986, pp. 322–331. IEEE, Piscataway (1986)
85. Wiedijk, F. (ed.): The Seventeen Provers of the World. LNCS, vol. 3600. Springer, Heidelberg (2006)
86. Wolper, P., Vardi, M.Y., Sistla, A.P.: Reasoning about infinite computation paths. In: Proceedings, Foundations of Computer Science, FOCS, Tucson, AZ, USA, November 7–9, 1983, pp. 185–194. IEEE, Piscataway (1983)
87. Yang, J., Hawblitzel, C.: Safe to the last instruction: automated verification of a type-safe operating system. Commun. ACM **54**(12), 123–131 (2011)

Chapter 2
Temporal Logic and Fair Discrete Systems

Nir Piterman and Amir Pnueli

Abstract Temporal logic has been used by philosophers to reason about the way the world changes over time. Its modern use in specification and verification of systems describes the evolution of states of a program/design giving rise to descriptions of executions. Temporal logics can be classified by their view of the evolution of time as either linear or branching. In the linear-time view, we see time ranging over a linear total order and executions are sequences of states. When the system has multiple possible executions (due to nondeterminism or reading input) we view them as separate possible evolutions and the system has a set of possible behaviors. In the branching-time view, a point in time may have multiple possible successors and accordingly executions are tree-like structures. According to this view, a system has exactly one execution, which takes the form of a tree. We start this chapter by introducing Fair Discrete Structures, the model on which we evaluate the truth and falsity of temporal logic formulas. Fair Discrete Structures describe the states of a system and their possible evolution. We then proceed with the linear-time view and introduce Propositional Linear Temporal Logic (LTL). We explain the distinction between safety and liveness properties and introduce a hierarchy of liveness properties of increasing expressiveness. We study the expressive power of full LTL and cover extensions that increase its expressive power. We introduce algorithms for checking the satisfiability of LTL and model checking LTL. We turn to the branching-time

Comment by Nir Piterman: Sadly, during the preparation of this manuscript, in November 2009, Amir passed away. While I tried to keep the chapter in the spirit of our plans I cannot be sure that Amir would have agreed with the final outcome. All faults and inaccuracies are definitely my own. In addition, opinions or judgements passed in this chapter may not reflect Amir's view. I would like to take this opportunity to convey my deep gratitude to Amir; my teacher, supervisor, mentor, and friend.

N. Piterman (✉)
University of Leicester, Leicester, UK
e-mail: nir.piterman@le.ac.uk

A. Pnueli
New York University, New York, USA

A. Pnueli
The Weizmann Institute of Science, Rehovot, Israel

E.M. Clarke et al. (eds.), *Handbook of Model Checking*,
DOI 10.1007/978-3-319-10575-8_2

framework and introduce Computation Tree Logic (CTL). As before, we discuss its expressive power, consider extensions, and cover satisfiability and model checking. We then dedicate some time to examples of formulas in both LTL and CTL and stress the differences between the two. We end with a formal comparison of LTL and CTL and, in view of this comparison, introduce CTL*, a hybrid of LTL and CTL that combines the linear and branching views into one logic.

2.1 Introduction

For many years, philosophers, theologians, and linguists have been using logic in order to reason about the world, freedom of choice, and the meaning of spoken language. In the middle of the twentieth century research on its foundations led to two complementary breakthroughs. In the late 1950s Arthur Prior introduced what we call today *tense logic*; essentially, introducing the operators Fp meaning "it will be the case that p" and its dual Gp meaning "it will always be the case that p" [44]. This work was done in the context of modal logic, where operators $\Box$ and $\Diamond$ meant "it must be" and "it is possible". Roughly at the same time, working on modal logic, Saul Kripke introduced a semantics, which we call today *Kripke semantics*, to interpret modal logic [28]. His suggestion was to interpret modal logic over a set of possible worlds and an accessibility relation between worlds. The modal operators are then interpreted over the accessibility relation of the worlds.

About 20 years later, these ideas penetrated the verification community. In a landmark paper, Pnueli introduced Prior's tense operators to verification and suggested that they can be used for checking the correctness of programs [42]. In particular, Pnueli's ideas considered ongoing and non-terminating computations of programs. The existing paradigm of verification was that of pre- and post-conditions matching programs that get an input and produce an output upon termination. Instead, what were later termed as reactive systems [23] continuously interact with their environment, receive inputs, send outputs, and importantly do not terminate. In order to describe the behaviors of such programs one needs to describe, for example, causality of interactions and their order. Temporal logic proved a convenient way to do this. It can be used to capture the specification of programs in a formal notation that can then be checked on programs.

Linear Temporal Logic (abbreviated LTL), the logic introduced by Pnueli, is linear in the sense that at every moment in time there is only one possible future. In the linear view an execution is a sequence of states. Multiple possible executions (e.g., due to different inputs) are treated separately as independent sequences. Logicians also had another possible option, to view time as branching: at every moment in time there may be multiple possible futures. An alternative view, that of branching time, was introduced to verification in [5]. In the branching-time approach, the program itself provides a branching structure, in which every possible snapshot of the program may have multiple options to continue. Then, the program is one logical structure that encompasses all possible behaviors. This was shortly followed by a second revolution. Clarke and Emerson [17] and Queille and Sifakis [45] introduced a more elaborate branching-time logic, Computation Tree Logic (abbreviated

CTL), and the idea of model checking: Model checking is the algorithmic analysis for determining whether a program satisfies a given temporal logic specification.

The linear–branching dichotomy, originating in a religious debate regarding freedom of choice and determinacy of fate, led to an ideological debate regarding the usage of temporal logic in verification (see [53]). Various studies compared the linear-time and branching-time approaches, which led also to the invention of CTL*, a logic that combines linear temporal logic and computation tree logic [19].

Over the years, extensive knowledge about all aspects of temporal logic and its usage in model checking and verification has been gathered: what properties can and cannot be expressed in various logics, how to extend the expressive power of logics, and algorithmic aspects of different questions. In particular, effective model-checking algorithms scaled to huge systems and established model checking as a successful technique in both industry and academia.

In this chapter, we cover the basics of temporal logic, the specification language used for model checking. In Sect. 2.2 we introduce Kripke semantics and a variant of it that we are going to use as a mathematical model for representing programs; we then present fair discrete systems and explain how to use them for representing programs. In Sect. 2.3 we present LTL, the basic linear temporal logic; we give the basic definition of the logic, discuss several possible extensions of it, and, finally, define and show how to perform model checking for LTL formulas. In Sect. 2.4 we present CTL, the basic branching temporal logic, including its definition, extensions to it, and model checking. Then, in Sect. 2.5 we cover examples for the usage of both logics. Finally, in Sect. 2.6, we compare the expressive power of the linear-time and branching-time variants and introduce the logic CTL*, a powerful branching-time logic that combines both LTL and CTL.

2.2 Fair Discrete Systems

In order to be able to formally reason about systems and programs, we have to agree on a formal mathematical model in which reasoning can be applied. Transition systems are, by now, a standard tool in computer science for representing programs. Transition systems are essentially labeled graphs, where labels appear either on states, edges, or both. There are many different variants of transition systems supplying many different flavors and supporting different needs. We define Kripke structures, one of the most popular versions of transition systems used in modeling. Then, we present a symbolic version of transition systems, which we call *fair discrete system*s or FDS for short, where the states arise as interpretations of variables and transitions correspond to changes in variables' values. As their name suggests, variables range over discrete domains, such as integers, Booleans, or other finite-range domains. Continuous variables will be considered later in this Handbook in [8] (Bouyer et al., Model Checking Real-Time Systems) and [14] (Doyen et al., Verification of Hybrid Systems). One of the most important features of programs and systems is concurrency, or the ability to communicate with other programs. Here, communication is by reading and writing the values of shared variables. In order to reason about multiple communicating programs (and also about temporal logic) our

systems include weak and strong fairness. Fairness requirements restrict our attention to certain paths in the system that "have some good qualities" when considering interaction and communication.

2.2.1 Kripke Structures

We give a short exposition of Kripke structures [28], one of the most popular formalisms for representing transition systems in the context of model checking. These are transition systems where states are elements of an abstract set, and initial states and transitions are defined over this explicit set. In addition, states are labeled with propositions (or observations about the state).

Definition 1 (Kripke structure) A Kripke structure is of the form $\mathscr{K} = \langle AP, S, S_0, R, L\rangle$, where AP is a finite set of atomic propositions, S is a set of states, $S_0 \subseteq S$ is a set of initial states, $R \subseteq S \times S$ is a transition relation, and $L : S \to 2^{AP}$ is a labeling function.

Kripke structures are state-labeled transition systems. We specify in advance a set of labels (propositions), which are the basic facts that might be known about the world. The labeling function associates every state with the set of atomic propositions that are true in it. The set of initial states and the transition relation allow us to add a notion of executions to Kripke structures.

Definition 2 (Paths and runs) A *path* of a Kripke structure $\mathscr{K}$ starting at state $s \in S$ is a maximal sequence of states $\sigma = s_0, s_1, \ldots$ such that $s_0 = s$ and for every $j \geq 0$ we have $(s_i, s_{i+1}) \in R$. A sequence σ is maximal if either σ is infinite or $\sigma = s_0, \ldots, s_k$ and s_k has no successor, i.e., for all $s \in S$ we have $(s_k, s) \notin R$.

A path starting in a state $s_0 \in S_0$ is called a *run*. We denote by $\mathsf{Runs}(\mathscr{K})$ the set of runs of $\mathscr{K}$.

The direct representation of states and transitions makes Kripke structures convenient for certain needs. However, in model checking, higher-level representations of transition systems provide a more direct relation to programs and enable us to reason about systems using sets of states rather than individual states. We now introduce one such modeling framework, called Fair Discrete Systems. The idea is that states are obtained by interpreting the values of variables of the system. In addition, in the context of model checking it is sometimes necessary to ignore some "not interesting" runs of the system. For this we introduce the notion of *fairness*, which is usually not considered with Kripke structures.

2.2.2 Definition of Fair Discrete System

Definition 3 (Fair discrete system) An FDS is of the form $\mathscr{D} = \langle \mathscr{V}, \theta, \rho, \mathscr{J}, \mathscr{C}\rangle$, where $\mathscr{V}$ is a finite set of typed variables, θ is an initial condition, ρ is a transition relation, $\mathscr{J}$ a set of justice requirements, and $\mathscr{C}$ a set of compassion requirements. Further details about these components are given below.

- $\mathcal{V} = \{v_1, \ldots, v_n\}$: A finite set of typed variables. Variables range over discrete domains, such as Booleans or integers. Although variables ranging over finite domains can be represented by multiple Boolean variables, sometimes it is convenient to use variables with a larger range. Program counters, ranging over the domain of program locations, are a prominent example.

  A *state* s is an interpretation of $\mathcal{V}$, i.e., if D_v is the domain of v, then s is an element in $\prod_{v_i \in \mathcal{V}} D_{v_i}$. We denote the set of possible states by $\Sigma_{\mathcal{V}}$. Given a subset $\mathcal{V}_1$ of $\mathcal{V}$, we denote by $s \Downarrow_{\mathcal{V}_1}$ the projection of s on the variables in $\mathcal{V}_1$. That is, the assignment to variables in $\mathcal{V}_1$ that agrees with s.

  When all variables range over finite domains, the system has a finite number of states, and is called a *finite-state system*. Otherwise, it is an *infinite-state system*.

  We assume some underlying first-order language over $\mathcal{V}$ that includes (i) *expressions* constructed from the variables in $\mathcal{V}$, (ii) *atomic formulas*, which are either Boolean variables or the application of different predicates to expressions, and (iii) *assertions*, which are first-order formulas constructed from atomic formulas using Boolean connectives or quantification of variables. Assertions, also sometimes called *state formulas*, characterize states through restriction of the possible variable values in them. For example, for a variable x ranging over integers, $x + 1 > 5$ is an atomic formula, and for a Boolean variable b, the assertion $\neg b \wedge x + 1 > 5$ characterizes the set of states where b is false and x is at least 5.
- θ: The *initial condition*. This is an assertion over $\mathcal{V}$ characterizing all the initial states of the FDS. A state is called *initial* if it satisfies θ.
- ρ: A *transition relation*. This is an assertion $\rho(\mathcal{V} \cup \mathcal{V}')$, where $\mathcal{V}'$ is a primed copy of the variables in $\mathcal{V}$. The transition relation ρ relates a state $s \in \Sigma$ to its $\mathcal{D}$*-successors* $s' \in \Sigma$, i.e., $(s, s') \models \rho$, where s supplies the interpretation to the variables in $\mathcal{V}$ and s' supplies the interpretation to the variables in $\mathcal{V}'$.

  For example, the assignment $x = x + 1$ is written $x' = x + 1$, stating that the next value of x is equal to the current value of x plus one. Given a set of variables $\mathcal{X} \subseteq \mathcal{V}$, we denote by $\mathsf{keep}(\mathcal{X})$ the formula $\bigwedge_{x \in \mathcal{X}} x = x'$, which preserves the values of all variables in $\mathcal{X}$.
- $\mathcal{J} = \{J_1, \ldots, J_m\}$: A set of *justice requirements* (weak fairness). Each requirement $J \in \mathcal{J}$ is an assertion over $\mathcal{V}$ that is intended to hold infinitely many times in every computation.
- $\mathcal{C} = \{(P_1, Q_1), \ldots, (P_n, Q_n)\}$: A set of *compassion requirements* (strong fairness). Each requirement $(P, Q) \in \mathcal{C}$ consists of a pair of assertions, such that if a computation contains infinitely many P-states, it should also contain infinitely many Q-states.

Definition 4 (Paths, runs, and computations) A *path* of an FDS $\mathcal{D}$ starting in state s is a maximal sequence of states $\sigma = s_0, s_1, \ldots$ such that $s_0 = s$ and for every $i \geq 0$ we have $(s_i, s'_{i+1}) \models \rho$, where s'_{i+1} is a primed copy of s_{i+1}, i.e., it is an assignment to $\mathcal{V}'$ such that for every $v \in \mathcal{V}$ we have $s_{i+1}(v) = s'_{i+1}(v')$. A sequence σ is maximal if either σ is infinite or $\sigma = s_0, \ldots, s_k$ and s_k has no $\mathcal{D}$-successor, i.e., for all $s_{k+1} \in \Sigma$, $(s_k, s'_{k+1}) \not\models \rho$. We denote by $|\sigma|$ the length of σ, that is $|\sigma| = \omega$ if

σ is infinite and $|\sigma| = k + 1$ if $\sigma = s_0, \ldots, s_k$ is finite. Given a path $\sigma = s_0, s_1, \ldots$, we denote by $\sigma \Downarrow_{\mathcal{V}_1}$ the path $s_0 \Downarrow_{\mathcal{V}_1}, s_1 \Downarrow_{\mathcal{V}_1}, \ldots$.

A path σ is *fair* if it is infinite and satisfies the following additional requirements: (i) *justice (or weak fairness)*, i.e., for each $J \in \mathcal{J}$, σ contains infinitely many J-positions, i.e., positions $j \geq 0$, such that $s_j \models J$, and (ii) *compassion (or strong fairness)*, i.e., for each $(P, Q) \in \mathcal{C}$, if σ contains infinitely many P-positions, it must also contain infinitely many Q-positions.

A path starting in a state s such that $s \models \theta$ is called a *run*. A fair path that is a run is called a *computation*. We denote by $\mathsf{Runs}(\mathcal{D})$ the set of runs of $\mathcal{D}$ and by $\mathsf{Comp}(\mathcal{D})$ the set of computations of $\mathcal{D}$.

A state s is said to be *reachable* if it appears in some run. It is *reachable from* t if it appears on some path starting in t. A state s is *viable* if it appears in some computation. An FDS is called *viable* if it has some computation.

An FDS is said to be *fairness-free* if $\mathcal{J} = \mathcal{C} = \emptyset$. It is called a *just discrete system* (JDS) if $\mathcal{C} = \emptyset$. When $\mathcal{J} = \emptyset$ or $\mathcal{C} = \emptyset$ we simply omit them from the description of $\mathcal{D}$. Note that for most reactive systems, it is sufficient to use a JDS (i.e., compassion-free) model. Compassion is only needed in cases in which the system uses built-in synchronization constructs such as semaphores or synchronous communication.

A fairness-free FDS can be converted to a Kripke structure. Given an FDS $\mathcal{D} = \langle \mathcal{V}, \theta, \rho \rangle$ and a set of basic assertions over its variables $\{a_1, \ldots, a_k\}$ the Kripke structure obtained from it is $\mathcal{K}_{\mathcal{D}} = \langle AP, S, S_0, R, L \rangle$, where the components of $\mathcal{K}_{\mathcal{D}}$ are as follows. The set of states S is the set of possible interpretations of the variables in $\mathcal{V}$, namely $\Sigma_{\mathcal{V}}$. The set of initial states S_0 is the set of states s such that $s \models \theta$, i.e., $S_0 = \{s \models \theta\}$. The transition relation R contains the pairs (s, t) such that $(s, t') \models \rho$. Notice that t' is a primed copy of t and is interpreted over the primed variables $\mathcal{V}'$. Finally, the set of propositions is $AP = \{a_1, \ldots, a_k\}$ and $a_i \in L(s)$ iff $s \models a_i$. We note that the number of states of the Kripke structure may be exponentially larger than the description of the FDS. We state without proof that this translation maintains the notion of a run. We note that one can add fairness to Kripke structures and define a notion of computation that is similar to that of FDSs.

Lemma 1 *Given an FDS $\mathcal{D}$, the sets of runs of $\mathcal{D}$ and $\mathcal{K}_{\mathcal{D}}$ are equivalent, namely,* $\mathsf{Runs}(\mathcal{D}) = \mathsf{Runs}(\mathcal{K}_{\mathcal{D}})$.

Sometimes it will be convenient to construct larger FDSs from smaller FDSs. Consider two FDSs $\mathcal{D}_1 = \langle \mathcal{V}_1, \theta_1, \rho_1, \mathcal{J}_1, \mathcal{C}_1 \rangle$ and $\mathcal{D}_2 = \langle \mathcal{V}_2, \theta_2, \rho_2, \mathcal{J}_2, \mathcal{C}_2 \rangle$, where $\mathcal{V}_1$ and $\mathcal{V}_2$ are not necessarily disjoint. The *synchronous parallel composition*, written $\mathcal{D}_1 ||| \mathcal{D}_2$, of $\mathcal{D}_1$ and $\mathcal{D}_2$ is the FDS defined as follows.

$$\mathcal{D}_1 ||| \mathcal{D}_2 = \langle \mathcal{V}_1 \cup \mathcal{V}_2, \theta_1 \wedge \theta_2, \rho_1 \wedge \rho_2, \mathcal{J}_1 \cup \mathcal{J}_2, \mathcal{C}_1 \cup \mathcal{C}_2 \rangle$$

A transition of the synchronous parallel composition is a joint transition of the two systems. A computation of the synchronous parallel composition when restricted to

Fig. 1 A simple loop

```
Var n:Integer initially n = 10
l0 : while (n > 0) {
l1 :    n = n − 1;
l2 : }
l3 :
```

the variables in one of the systems is a computation of that system. Formally, we have the following.

Lemma 2 *A sequence* $\sigma \in (\Sigma_{\mathscr{V}_1 \cup \mathscr{V}_2})^\omega$ *is a computation of* $\mathscr{D}_1 \,|||\, \mathscr{D}_2$ *iff* $\sigma \Downarrow_{\mathscr{V}_1}$ *is a computation of* $\mathscr{D}_1$ *and* $\sigma \Downarrow_{\mathscr{V}_2}$ *is a computation of* $\mathscr{D}_2$.

The proof is omitted.

The *asynchronous parallel composition* $\mathscr{D}_1 \| \mathscr{D}_2$ of $\mathscr{D}_1$ and $\mathscr{D}_2$ is the FDS defined as follows.

$$\mathscr{D}_1 \| \mathscr{D}_2 = \langle \mathscr{V}_1 \cup \mathscr{V}_2, \theta_1 \wedge \theta_2, \rho, \mathscr{J}_1 \cup \mathscr{J}_2, \mathscr{C}_1 \cup \mathscr{C}_2 \rangle,$$

where $\rho = (\rho_1 \wedge \mathsf{keep}(\mathscr{V}_2 \setminus \mathscr{V}_1)) \vee (\rho_2 \wedge \mathsf{keep}(\mathscr{V}_1 \setminus \mathscr{V}_2))$. A transition of the asynchronous parallel composition is a transition of one of the systems preserving unchanged the variables of the other. A computation of the asynchronous parallel composition, when restricted to the variables in one of the systems is not necessarily a computation of that system. For example, if the sets of variables of the two systems intersect and system one modifies the variables of system two, the projection of the computation on the variables of system two could include changes not allowed by the transition of system two.

2.2.3 Representing Programs

We show how FDSs can represent programs. We do not formally define a programming language, however, the meaning of commands and constructs will be clear from the translation to FDSs. A more thorough discussion of representation of programs is given elsewhere in this Handbook in [47] (Seshia et al., Modeling for Verification). FDSs are a simple variant of the State Transition Systems (STSs) defined in that chapter.

Consider for example the program in Fig. 1. It can be represented as an FDS with the variables π and n, where π is the program location variable ranging over $\{l_0, \ldots, l_3\}$ and n is an integer that starts as 10. Formally, $\mathscr{D} = \langle \{\pi, n\}, \theta, \rho, \mathscr{J}, \mathscr{C} \rangle$, where $\mathscr{J} = \emptyset$, $\mathscr{C} = \emptyset$, and θ and ρ are as follows.

$$\begin{aligned}
\theta &: \pi = l_0 \wedge n = 10, \\
\rho &: (\pi = l_0 \wedge n > 0 \wedge \pi' = l_1 \wedge n' = n) \vee (\pi = l_0 \wedge n \leq 0 \wedge \pi' = l_3 \wedge n' = n) \vee \\
&\quad (\pi = l_1 \wedge \pi' = l_2 \wedge n' = n - 1) \quad \vee (\pi = l_2 \wedge \pi' = l_0 \wedge n' = n) \quad \vee \\
&\quad (\pi' = \pi \wedge n' = n).
\end{aligned}$$

For software programs, we always assume that the transition relation ρ includes as a disjunct the option to *stutter*, that is, do nothing. This allows us to model the environment of a single processor that devotes attention to one of many threads. Given a program counter variable, we denote by at_{l_i} the formula $\pi = l_i$. In case of multiple program counters, we assume that their ranges are disjoint and identify the right variable by its range, e.g., one program counter ranges over l_i and the other over m_i making $\pi = m_i$ unambiguous. Similarly, at'_{l_i} is $\pi' = l_i$

The following sequence of states is a run of the simple loop in Fig. 1.

$$\begin{aligned}\sigma = {} & \langle \pi : l_0, n : 10\rangle, \langle \pi : l_1, n : 10\rangle, \langle \pi : l_2, n : 9\rangle, \langle \pi : l_0, n : 9\rangle, \langle \pi : l_1, n : 9\rangle, \\ & \langle \pi : l_2, n : 8\rangle, \langle \pi : l_0, n : 8\rangle, \ldots, \langle \pi : l_0, n : 1\rangle, \langle \pi : l_1, n : 1\rangle, \langle \pi : l_2, n : 0\rangle, \\ & \langle \pi : l_0, n : 0\rangle, \langle \pi : l_3, n : 0\rangle, \ldots\end{aligned}$$

Indeed, it starts in an initial state, where $\pi = l_0$ and $n = 10$, and every two adjacent states satisfy the transition relation. For example, the two states $\langle \pi : l_1, n : 9\rangle$ and $\langle \pi : l_2, n : 8\rangle$ satisfy the disjunct $\mathsf{at}_{l_1} \wedge \mathsf{at}'_{l_2} \wedge n' = n - 1$ where π and n range over the first state and π' and n' range over the second state.

Consider the two processes in Fig. 2, depicting Peterson's mutual exclusion algorithm [41]. Consider the process on the left. It can be represented as an FDS with Boolean variables x, y, and t and a location variable π_1 ranging over $\{l_0, \ldots, l_7\}$. Formally, $\mathscr{D}_1 = \langle \{\pi_1, x, y, t\}, \theta_1, \rho_1, \mathscr{J}, \mathscr{C}\rangle$, where the components of $\mathscr{D}_1$ are as follows.

$$\begin{aligned}\theta_1 \ : {} & \mathsf{at}_{l_0} \wedge x = 0, \\ \rho_1 \ : {} & (\mathsf{at}_{l_0} \wedge \mathsf{at}'_{l_1} \wedge \mathsf{keep}(x, y, t)) \quad \vee (\mathsf{at}_{l_1} \wedge \mathsf{at}'_{l_2} \wedge \mathsf{keep}(x, y, t)) \quad \vee \\ & (\mathsf{at}_{l_2} \wedge \mathsf{at}'_{l_3} \wedge x' = 1 \wedge \mathsf{keep}(y, t)) \vee (\mathsf{at}_{l_3} \wedge \mathsf{at}'_{l_4} \wedge t' = 1 \wedge \mathsf{keep}(x, y)) \vee \\ & (\mathsf{at}_{l_4} \wedge (t = 0 \vee y = 0) \wedge \mathsf{at}'_{l_5} \wedge \mathsf{keep}(x, y, t)) \quad \vee \\ & (\mathsf{at}_{l_5} \wedge \mathsf{at}'_{l_6} \wedge \mathsf{keep}(x, y, t)) \quad \vee (\mathsf{at}_{l_6} \wedge x' = 0 \wedge \mathsf{at}'_{l_7} \wedge \mathsf{keep}(y, t)) \vee \\ & (\mathsf{at}_{l_7} \wedge \mathsf{at}'_{l_0} \wedge \mathsf{keep}(x, y, t)) \quad \vee \mathsf{keep}(\pi_1, x, y, t).\end{aligned}$$

Notice that the disjunct $\mathsf{keep}(\pi_1, x, y, t)$ allows this process to stutter, but also includes the transition from l_4 to l_4 in case $t \neq 0$ and $y \neq 0$. Dually, the process on the right can be represented as an FDS with Boolean variables $\{x, y, t\}$ and location variable π_2 ranging over $\{m_0, \ldots, m_7\}$. Formally $\mathscr{D}_2 = \langle \{\pi_2, x, y, t\}, \theta_2, \rho_2, \mathscr{J}, \mathscr{C}\rangle$, where the components of $\mathscr{D}_2$ are as follows.

$$\begin{aligned}\theta_2 \ : {} & \mathsf{at}_{m_0} \wedge y = 0, \\ \rho_2 \ : {} & (\mathsf{at}_{m_0} \wedge \mathsf{at}'_{m_1} \wedge \mathsf{keep}(x, y, t)) \quad \vee (\mathsf{at}_{m_1} \wedge \mathsf{at}'_{m_2} \wedge \mathsf{keep}(x, y, t)) \quad \vee \\ & (\mathsf{at}_{m_2} \wedge \mathsf{at}'_{m_3} \wedge y' = 1 \wedge \mathsf{keep}(x, t)) \vee (\mathsf{at}_{m_3} \wedge \mathsf{at}'_{m_4} \wedge t' = 0 \wedge \mathsf{keep}(x, y)) \vee \\ & (\mathsf{at}_{m_4} \wedge (t = 1 \vee x = 0) \wedge \mathsf{at}'_{m_5} \wedge \mathsf{keep}(x, y, t)) \quad \vee \\ & (\mathsf{at}_{m_5} \wedge \mathsf{at}'_{m_6} \wedge \mathsf{keep}(x, y, t)) \quad \vee (\mathsf{at}_{m_6} \wedge y' = 0 \wedge \mathsf{at}'_{m_7} \wedge \mathsf{keep}(x, t)) \vee \\ & (\mathsf{at}_{m_7} \wedge \mathsf{at}'_{m_0} \wedge \mathsf{keep}(x, y, t)) \quad \vee \mathsf{keep}(\pi_2, x, y, t).\end{aligned}$$

In addition we add an FDS $\mathscr{T}$ that sets $t = 0$ initially. Let $\mathscr{T} = \langle \{t\}, t = 0, t = t', \emptyset, \emptyset\rangle$.

$$\texttt{Var } t,x,y\texttt{: Boolean initially } t=0,\ x=0,\ y=0$$

$$\begin{bmatrix} l_0 : \texttt{while (true) \{} \\ l_1 : \quad \texttt{Non Critical;} \\ l_2 : \quad x=1; \\ l_3 : \quad t=1; \\ l_4 : \quad \texttt{await } (t==0 \vee y==0); \\ l_5 : \quad \texttt{Critical;} \\ l_6 : \quad x=0; \\ l_7 : \texttt{\}} \end{bmatrix} \parallel \begin{bmatrix} m_0 : \texttt{while (true) \{} \\ m_1 : \quad \texttt{Non Critical;} \\ m_2 : \quad y=1; \\ m_3 : \quad t=0; \\ m_4 : \quad \texttt{await } (t==1 \vee x==0); \\ m_5 : \quad \texttt{Critical;} \\ m_6 : \quad y=0; \\ m_7 : \texttt{\}} \end{bmatrix}$$

Fig. 2 Peterson's mutual exclusion algorithm

In order to obtain an FDS that represents the entire behavior of the two processes together, we construct the asynchronous parallel composition of $\mathscr{D}_1$, $\mathscr{D}_2$, and $\mathscr{T}$. That is, the FDS representing Peterson's mutual exclusion protocol is $\mathscr{D} = \mathscr{D}_1 \| \mathscr{D}_2 \| \mathscr{T}$.

The following sequence of states is a run of the processes in Fig. 2. Between every two states we write the location from which a transition is applied.

$$\begin{aligned} \sigma : \langle \pi_1 : l_0, x:0, \pi_2 : m_0, y:0, t:0\rangle &\xrightarrow{m_0} \langle \pi_1 : l_0, x:0, \pi_2 : m_1, y:0, t:0\rangle \xrightarrow{m_1} \\ \langle \pi_1 : l_0, x:0, \pi_2 : m_2, y:0, t:0\rangle &\xrightarrow{m_2} \langle \pi_1 : l_0, x:0, \pi_2 : m_3, y:1, t:0\rangle \xrightarrow{l_0} \\ \langle \pi_1 : l_1, x:0, \pi_2 : m_3, y:1, t:0\rangle &\xrightarrow{l_1} \langle \pi_1 : l_2, x:0, \pi_2 : m_3, y:1, t:0\rangle \xrightarrow{l_2} \\ \langle \pi_1 : l_3, x:1, \pi_2 : m_3, y:1, t:0\rangle &\xrightarrow{l_3} \langle \pi_1 : l_4, x:1, \pi_2 : m_3, y:1, t:1\rangle \xrightarrow{l_4} \\ \langle \pi_1 : l_4, x:1, \pi_2 : m_3, y:1, t:1\rangle &\xrightarrow{l_4} \cdots \xrightarrow{l_4} \cdots \xrightarrow{l_4} \cdots \xrightarrow{l_4} \cdots \xrightarrow{l_4} \cdots \end{aligned}$$

However, this run seems to violate our basic intuition regarding scheduling of different threads. Indeed, from some point onwards the processor gives attention only to the first thread. In order to remove such behaviors we add the following two justice requirements.

$$\begin{aligned} \mathscr{J}_1 &: \left\{ \neg \mathsf{at}_{l_i}, \neg \mathsf{at}_{l_4} \vee (t=1 \wedge y=1) \;\middle|\; i \in \{0,2,3,5,6,7\} \right\}, \\ \mathscr{J}_2 &: \left\{ \neg \mathsf{at}_{m_i}, \neg \mathsf{at}_{m_4} \vee (t=0 \wedge x=1) \;\middle|\; i \in \{0,2,3,5,6,7\} \right\}. \end{aligned}$$

The justice requirement for their composition $\mathscr{D}$ is $\mathscr{J} = \mathscr{J}_1 \cup \mathscr{J}_2$. Notice that it is fine for the processes to remain forever in their non-critical sections. Now, the run above violates the justice requirement. Indeed, the requirement $\neg \mathsf{at}_{m_3}$ does not hold on the last state. Thus, in this run there are only finitely many positions satisfying $\neg \mathsf{at}_{m_3}$. In order to constitute a computation the following suffix, for example, could be added.

$$\begin{aligned} \langle \pi_1 : l_4, x:1, \pi_2 : m_3, y:1, t:1\rangle &\xrightarrow{m_3} \langle \pi_1 : l_4, x:1, \pi_2 : m_4, y:1, t:0\rangle \xrightarrow{l_4} \\ \langle \pi_1 : l_5, x:1, \pi_2 : m_4, y:1, t:0\rangle &\xrightarrow{l_5} \langle \pi_1 : l_6, x:1, \pi_2 : m_4, y:1, t:0\rangle \xrightarrow{l_6} \\ \langle \pi_1 : l_7, x:0, \pi_2 : m_4, y:1, t:0\rangle &\xrightarrow{m_4} \langle \pi_1 : l_7, x:0, \pi_2 : m_5, y:1, t:0\rangle \cdots \end{aligned}$$

```
Var x: Boolean initially x = 1

[ l0 : while (true) {        ]   [ m0 : while (true) {        ]
[ l1 :   Non Critical;       ]   [ m1 :   Non Critical;       ]
[ l2 :   request (x);        ] ‖ [ m2 :   request (x);        ]
[ l3 :   Critical;           ]   [ m3 :   Critical;           ]
[ l4 :   release (x);        ]   [ m4 :   release (x);        ]
[ l5 : }                     ]   [ m5 : }                     ]
```

Fig. 3 Mutual exclusion using semaphores

We now turn to an example eliciting the need for compassion. Consider the two processes in Fig. 3. Here, the command `request`(x) is an atomic statement equivalent to `await`$(x == 1); x = 0$; Thus, at locations l_2 and m_2 the respective process proceeds only if x is 1 and it sets x to 0. The command `release`(x) sets x to 1 again. The process on the left can be represented as an FDS with a Boolean variable x and a location variable π_1 ranging over $\{l_0, \ldots, l_5\}$. Formally $\mathscr{D}_1 = \langle\{\pi_1, x\}, \theta_1, \rho_1, \mathscr{J}, \mathscr{C}\rangle$, where the components of $\mathscr{D}_1$ are as follows.

$$
\begin{aligned}
\theta_1 \ &: \mathsf{at}_{l_0} \wedge x = 1 \\
\rho_1 \ &: (\mathsf{at}_{l_0} \wedge \mathsf{at}'_{l_1} \wedge \mathsf{keep}(x)) \qquad \vee (\mathsf{at}_{l_1} \wedge \mathsf{at}'_{l_2} \wedge \mathsf{keep}(x)) \vee \\
&\quad (\mathsf{at}_{l_2} \wedge \mathsf{at}'_{l_3} \wedge x = 1 \wedge x' = 0) \vee (\mathsf{at}_{l_3} \wedge \mathsf{at}'_{l_4} \wedge \mathsf{keep}(x)) \vee \\
&\quad (\mathsf{at}_{l_4} \wedge \mathsf{at}'_{l_5} \wedge x' = 1) \qquad \vee (\mathsf{at}_{l_5} \wedge \mathsf{at}'_{l_0} \wedge \mathsf{keep}(x)) \vee \mathsf{keep}(\pi_1, x)
\end{aligned}
$$

The system $\mathscr{D}_2$ is obtained from $\mathscr{D}_1$ by replacing every reference to l by a reference to m. It is clear that we also have to include requirements from a scheduler. This time, location l_2 is problematic. Suppose that we try, as before, the following sets.

$$
\begin{aligned}
\mathscr{J}_1 &: \left\{\neg\mathsf{at}_{l_i}, \neg\mathsf{at}_{l_2} \vee x = 0 \;\middle|\; i \in \{0, 3, 4, 5\}\right\}, \\
\mathscr{J}_2 &: \left\{\neg\mathsf{at}_{m_i}, \neg\mathsf{at}_{m_2} \vee x = 0 \;\middle|\; i \in \{0, 3, 4, 5\}\right\}.
\end{aligned}
$$

However, it is simple to see that this is not strong enough. Consider the following computation.

$$
\begin{aligned}
\sigma : \ & \langle \pi_1 : l_0, \pi_2 : m_0, x : 1\rangle \xrightarrow{m_0} \langle \pi_1 : l_0, \pi_2 : m_1, x : 1\rangle \xrightarrow{m_1} \\
& \langle \pi_1 : l_0, \pi_2 : m_2, x : 1\rangle \xrightarrow{m_2} \langle \pi_1 : l_0, \pi_2 : m_3, x : 0\rangle \xrightarrow{l_0} \\
& \langle \pi_1 : l_1, \pi_2 : m_3, x : 0\rangle \xrightarrow{l_1} \langle \pi_1 : l_2, \pi_2 : m_3, x : 0\rangle \xrightarrow{l_2} \\
& \langle \pi_1 : l_2, \pi_2 : m_3, x : 0\rangle \xrightarrow{m_3} \langle \pi_1 : l_2, \pi_2 : m_4, x : 0\rangle \xrightarrow{m_4} \\
& \langle \pi_1 : l_2, \pi_2 : m_5, x : 1\rangle \xrightarrow{m_5} \cdots \xrightarrow{m_0} \cdots \xrightarrow{m_1} \cdots \xrightarrow{m_2} \cdots \xrightarrow{l_2} \cdots \xrightarrow{m_3} \cdots \xrightarrow{m_4} \cdots
\end{aligned}
$$

This computation keeps $\mathscr{D}_1$ in location 2 forever. This is indeed a computation as there are infinitely many positions where the transition from l_2 is not enabled and $\neg\mathsf{at}_{l_2} \vee x = 0$ holds. Clearly, this does not seem acceptable. One could ask why not replace the justice requirement related to location 2 by $\neg\mathsf{at}_{l_2}$. However, this is too strong. If we replace $\mathscr{D}_2$ by an FDS that sets x to 0 and never resets it to 1, clearly, we cannot expect a computation that fulfils the justice requirement $\neg\mathsf{at}_{l_2}$.

In order to solve this problem we replace the justice requirements relating to location 2 by the following compassion.

$$\mathscr{C}_1 : \{\langle \mathsf{at}_{l_2} \wedge x = 1, \neg \mathsf{at}_{l_2} \rangle\},$$
$$\mathscr{C}_2 : \{\langle \mathsf{at}_{m_2} \wedge x = 1, \neg \mathsf{at}_{m_2} \rangle\}.$$

Namely, if there are enough opportunities where $\mathscr{D}_1$ is at location 2 and x is free, we expect $\mathscr{D}_1$ to get an option to move when x is free; and similarly for $\mathscr{D}_2$.

2.2.4 Algorithms

We now consider the algorithms that show whether a given FDS has some computation. For infinite-state systems the question is in general undecidable, as an infinite-state system can easily represent the halting problem or its dual. For various kinds of infinite-state systems this question is discussed in later chapters of this Handbook [25] (Jhala et al., Predicate Abstraction for Program Verification), [8], and [14]. Here, we concentrate on the case of finite-state systems, where all variables range over finite domains. For such systems, simple algorithms establish whether an FDS is viable. We start with the simple case of a fairness-free FDS. We then show how to solve the same problem for JDSs and for general FDSs. Here, we concentrate on symbolic algorithms that handle sets of states. We assume that we have some efficient way to represent, manipulate, and compare assertions. One such system is discussed elsewhere in this Handbook [9] (Bryant, Binary Decision Diagrams). Enumerative algorithms, which consider individual states, are discussed in this Handbook in [24] (Holzmann, Explicit-State Model Checking).

We fix an FDS $\mathscr{D} = \langle \mathscr{V}, \theta, \rho, \mathscr{J}, \mathscr{C} \rangle$. In the algorithm we use the operators $\mathsf{post}()$ and $\mathsf{pre}()$. The operator $\mathsf{post}(S, \rho)$ returns the set $\{t \mid \exists s \in S \,.\, (s, t') \models \rho\}$ of successors of the states in S according to ρ. Similarly, the operator $\mathsf{pre}(S, \rho)$ returns the set $\{s \mid \exists t \in S \,.\, (s, t') \models \rho\}$ of predecessors of the states in S. For an assertion τ characterizing the set of states S, we use the following operators.

$$\mathsf{prime}(\tau) = \exists \mathscr{V} \,.\, (\tau \wedge (\textstyle\bigwedge_{v \in \mathscr{V}} v = v')) \qquad \mathsf{unprime}(\tau) = \exists \mathscr{V}' \,.\, (\tau \wedge (\textstyle\bigwedge_{v \in \mathscr{V}} v = v'))$$
$$\mathsf{post}(\tau, \rho) = \mathsf{unprime}(\exists \mathscr{V} \,.\, (\tau \wedge \rho)) \qquad \mathsf{pre}(\tau, \rho) = \exists \mathscr{V}' \,.\, (\mathsf{prime}(\tau) \wedge \rho)$$

For example, given an assertion τ, the operator $\mathsf{post}(\tau, \rho)$ produces an assertion describing the successors of the states satisfying τ. Indeed, the assertion $\tau \wedge \rho$ describes the pairs of states satisfying the transition relation ρ such that the first (unprimed) satisfies τ. Then, $\exists \mathscr{V} \,.\, (\tau \wedge \rho)$ quantifies out the first state leaving us with a description over the set of variables $\mathscr{V}'$ of the set of successors of states in τ. Finally, $\mathsf{unprime}(\exists \mathscr{V} \,.\, (\tau \wedge \rho))$ converts it to an assertion over the variables in $\mathscr{V}$. The operator $\mathsf{pre}(\tau, \rho)$ first converts the assertion τ to an assertion over $\mathscr{V}'$ using the operator $\mathsf{prime}(\tau)$, then the conjunction with ρ creates a description of pairs that satisfy the transition such that the second state (primed) satisfies τ. Finally, quantifying the primed variables creates an assertion describing the predecessors of τ.

Algorithm 1 Reachable states

```
1: new := θ;
2: old := ¬new;
3: while (new ≠ old)
4:     old := new;
5:     new := new ∨ post(new, ρ)
6: end while
```

We are now ready to present the first algorithm. Algorithm 1 computes an assertion characterizing the reachable states. All variables range over assertions.

As the system has a finite number of states, it is clear that this algorithm terminates. Indeed, each time the loop body is run the assertion new characterizes more and more states. As the number of states is finite, at some point, old is equivalent to new. It is also simple to see that when the loop terminates, the assertion new characterizes the set of reachable states. Indeed, the algorithm starts with all initial states and gradually adds states that are reachable with an increasing number of steps. We do not prove this formally. To simplify future algorithms, we introduce the operators reach(τ, ρ) and backreach(τ, ρ) that compute the set of states reachable from τ using the transition relation ρ and the set of states that can reach τ using the transition relation ρ, respectively. The operator reach(τ, ρ) is the result of running the algorithm for reachability initializing new to τ. The operator backreach(τ, ρ) is the result of running the algorithm obtained from the algorithm for reachability by replacing the usage of post() by pre() and initializing new to τ.

Algorithm 1 computes the fixpoint of the operator new := new $\vee$ post(new, ρ) in the loop body. We introduce a shorthand for this type of **while** loop. The loop header **fix**(new:= τ), initializes the variable new to τ, initializes the variable old to $\neg\tau$, terminates when old and new represent the same assertion, and updates old to new whenever it starts the loop body. We denote by new_0 the initial value of the loop variable and by new_i the value at the end of the ith iteration. Generally, as for some $j \geq 0$ we have $\mathsf{new}_{j+1} = \mathsf{new}_j$, we consider the value of new_i for $i \geq j$ to be new_j. Let $\mathsf{new}_{\text{fix}}$ denote the value of the variable new when exiting the loop.

We now turn our attention to the question of viability, starting in the case of fairness-free FDSs.

For finite-state FDS, viability is essentially reduced to finding the set of reachable states from which infinite paths can start.

Again, like the reachability algorithm, it is clear that this algorithm terminates. The set of states represented by new is non-increasing when recomputed by the loop body. As the system is finite-state, at some point, this set either becomes empty or does not change, leading to termination.

Lemma 3 *For a fairness free FDS $\mathscr{D}$ Algorithm 2 computes the set of viable states.*

Proof We show that for every i and for every s such that $s \models \mathsf{new}_i$ there is a path of length at least $i + 1$ starting from s. For $i = 0$ this clearly holds as from every

Algorithm 2 Viability (no fairness)

```
1:  reach := reach(θ, ρ);
2:  fix (new := reach)
3:      ρ_f := ρ ∧ new ∧ prime(new);
4:      new := new ∧ pre(new, ρ_f);
13: end fix
```

state there is a path of length 1. Suppose that for $s \models \mathsf{new}_i$ there is a path of length at least $i+1$ starting at s. Then clearly, $\mathsf{new}_{i+1} = \mathsf{new}_i \wedge \mathsf{pre}(\mathsf{new}_i, \rho)$ includes the set of states from which there is a path of length at least $i+2$. It follows that for every $s \models \mathsf{new}_{\mathrm{fix}}$ for every $i \geq 0$ there is a path of length $i+1$ starting at s. Arrange all the paths starting in s in the form of a tree such that a path of length $i+1$ is a descendant of a path of length i. The tree has finite branching degree and an infinite number of nodes. It follows from König's lemma that the tree contains an infinite path. Hence, from s there is an infinite path.

Let inf denote the set of reachable states s that have an infinite path starting from them. Then, $\mathrm{inf} \rightarrow \mathsf{new}_{\mathrm{fix}}$. Indeed, for every $i \geq 0$ we have $\mathrm{inf} \rightarrow \mathsf{new}_i$. Clearly, $\mathrm{inf} \rightarrow \mathsf{new}_0$. Suppose that $\mathrm{inf} \rightarrow \mathsf{new}_i$. But for every state s such that $s \models \mathrm{inf}$ there is a successor t such that $t \models \mathrm{inf}$. Indeed, this state t is the first state after s in the infinite path starting from s. Then, for every $s \models \mathrm{inf}$ we have $s \models \mathsf{new}_{i+1}$.

Finally, as new starts from the set of reachable states, if new is not empty the FDS has some infinite run. □

If a system contains the stuttering clause, i.e., every state is allowed to stutter, Algorithm 2 returns all reachable states. Indeed, for every state s we have $(s, s) \models \rho$. We start with new including all reachable states. Then, $\mathsf{pre}(\mathsf{new}, \rho_f)$ is new again. So the fixpoint terminates immediately with all states. This is because in such system every state has a computation that remains forever in that state. In order to restrict attention to "interesting" infinite computations, we could replace the transition relation ρ in Algorithm 2 by $\rho \wedge \bigvee_{v \in \mathscr{V}} v \neq v'$ that removes stuttering steps from ρ.

We now turn to consider JDSs. For such systems, it is not enough to just find infinite paths in the system. We have to ensure in addition that we can find an infinite path that visits each $J \in \mathscr{J}$ infinitely often. For that, we extend Algorithm 2 by adding the lines 5–8.

Lemma 4 *For a JDS $\mathscr{D}$ Algorithm 3 computes the set of viable states.*

Proof Every state s such that $s \models \mathsf{new}_{\mathrm{fix}}$ is reachable. Furthermore, for every s such that $s \models \mathsf{new}_{\mathrm{fix}}$ and for every $J \in \mathscr{J}$ we have that there is some state t reachable from s such that $t \models \mathsf{new}_{\mathrm{fix}}$, $t \models J$ and t has a successor in $\mathsf{new}_{\mathrm{fix}}$. It is possible now to construct paths of increasing length that visit all $J \in \mathscr{J}$. Let $\mathscr{J} = \{J_1, \ldots, J_m\}$. Start from some state s_0 in $\mathsf{new}_{\mathrm{fix}}$. Then, there is a state t_1 in $\mathsf{new}_{\mathrm{fix}}$ reachable from s that satisfies J_1 with a successor s_1 in $\mathsf{new}_{\mathrm{fix}}$. Similarly, extend this path from s_0 to

Algorithm 3 Viability (justice)

```
 1: reach := reach(θ, ρ);
 2: fix (new := reach)
 3:     ρ_f := ρ ∧ new ∧ prime(new);
 4:     new := new ∧ pre(new, ρ_f);
 5:     for all (J ∈ 𝒥)
 6:         reach_J := backreach(new ∧ J, ρ_f);
 7:         new := new ∧ reach_J;
 8:     end for {all (J ∈ 𝒥)}
13: end fix
```

Algorithm 4 Viability (compassion)

```
 1: reach := reach(θ, ρ);
 2: fix (new := reach)
 3:     ρ_f := ρ ∧ new ∧ prime(new);
 4:     new := new ∧ pre(new, ρ_f);
 5:     for all (J ∈ 𝒥)
 6:         reach_J := backreach(new ∧ J, ρ_f);
 7:         new := new ∧ reach_J;
 8:     end for {all (J ∈ 𝒥)}
 9:     for all ((P, Q) ∈ 𝒞)
10:         reach_Q := backreach(new ∧ Q, ρ_f);
11:         new := new ∧ (¬P ∨ reach_Q);
12:     end for {all ((P, Q) ∈ 𝒞)}
13: end fix {(new)}
```

s_1 to a path that visits also J_2 and ends in s_2 and so on. As before, we organize these paths in the form of a tree. An infinite path in this tree visits all $J \in \mathcal{J}$ infinitely often. Indeed, such an infinite path is the limit of paths that visit all $J \in \mathcal{J}$ an increasing number of times.

In the other direction we show that every viable state appears in $\mathsf{new}_{\mathrm{fix}}$. Similarly to the previous proof, a computation that starts from state s is used to show that s can reach all $J \in \mathcal{J}$. The suffix of this path is used to show that it is possible to continue from s to some successor in the fixpoint. □

Finally, we consider a general FDS. We extend Algorithm 3 by adding the lines 9–12. These lines ensure, in addition, that every compassion requirement is satisfied. This algorithm is due to [27].

Lemma 5 *For an FDS $\mathcal{D}$ Algorithm 4 computes the set of viable states.*

The proof is similar to that of Lemma 4.

Theorem 1 *Viability of a finite-state FDS can be checked in time polynomial in the number of states of the FDS and space logarithmic in the number of states.*

The polynomial upper bound follows from the analysis of termination of the algorithm. The space complexity follows from the algorithms in [54].

We note that the number of states of the system may be exponential in the size of its description. Indeed, a system with n Boolean variables potentially has 2^n states. Thus, if we consider the size of the description of the FDS, these algorithms can be considered to run in exponential time and polynomial space, respectively.

The reliance of these algorithms on computing the set of reachable states makes it problematic to apply them to infinite-state systems. Indeed, for such systems the iterative search for new states may never terminate. For some infinite-state systems, other techniques for the computation of the reachable and back-reachable states are available. In such cases, viability algorithms may be available. See elsewhere in this Handbook [8], [14], and [2] (Alur et al., Model Checking Procedural Programs) for treatment of infinite-state systems.

In what follows we use FDSs as a formal model of systems. We now turn our attention to a way to describe properties of such systems. We start with a description of linear temporal logic.

2.3 Linear Temporal Logic

We use logic to specify computations of FDS. Specifications that tell us what computations should and should not do are written formally as logic formulas. The temporal operators of the logic connect different stages of the computation and talk about dependencies and relations between them. Then, correctness of computations can be checked by checking whether they satisfy logical specifications. Here, we present linear temporal logic (abbreviated LTL). As its name suggests, it takes the linear-time view. That is, every point in time has a unique successor and models are infinite sequences of "worlds". Systems, as defined in Sect. 2.2, may have multiple successors for a given state. From LTL's point of view, a branching from some state would result in two different computations that are considered separately (though they share a prefix). We start with a definition of LTL, viewing infinite sequences as models of the logic and a logical formula as defining a set of models. We then discuss several extensions of LTL and what properties it can (and cannot) express. Finally, we connect the discussion to the FDSs defined in Sect. 2.2, define model checking, and show how to perform it.

We note that the discussion of LTL is restricted to infinite models. When considering FDS $\mathscr{D}$, we assume that $\mathsf{Runs}(\mathscr{D})$ contains only infinite paths. If this is not the case, some modifications need to be made either to the definition of LTL or the FDS itself in order to handle finite paths, see later in this Handbook [15] (Eisner and Fisman, Functional Specification of Hardware via Temporal Logic).

2.3.1 Definition of Linear Temporal Logic

We assume a countable set of Boolean propositions P. A model σ for a formula φ is an infinite sequence of truth assignments to propositions. Namely, if $\hat{P}$ is the set of propositions appearing in φ, then for every finite set P such that $\hat{P} \subseteq P$, a word in $(2^P)^\omega$ is a model. Given a model $\sigma = \sigma_0, \sigma_1, \ldots$, we denote by σ_i the set of propositions at position i. We also refer to sets of models as *languages*.

LTL formulas are constructed using the normal Boolean connectives for disjunction and negation and introduce the temporal operators *next*, *previous*, *until*, and *since*.

$$\varphi ::= p \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \bigcirc \varphi \mid \ominus \varphi \mid \varphi_1 \mathscr{U} \varphi_2 \mid \varphi_1 \mathscr{S} \varphi_2. \tag{1}$$

As mentioned, a model for an LTL formula is an infinite sequence. While Boolean connectives have their 'normal roles', the temporal connectives go between the different locations in the model. Intuitively, the unary *next* operator $\bigcirc$ indicates that the rest of the formula is true in the next location in the sequence; the unary *previous* operator $\ominus$ indicates that the rest of the formula is true in the previous location; the binary *until* operator $\mathscr{U}$ indicates that its first operand holds at all points in the future until some future point where its second operand holds; and dually (with respect to time) the binary *since* operator $\mathscr{S}$ indicates that its first operand holds at all points in the past until some past point where its second operand holds. Formulas that do not use $\ominus$ or $\mathscr{S}$ are *pure future* formulas. Formulas that do not use $\bigcirc$ or $\mathscr{U}$ are *pure past* formulas. In many cases (in practice and in this Handbook), attention is restricted to pure future LTL and the past is omitted.

In general, as we will see below the satisfaction of a formula is considered at a certain position of a model. If we are interested in satisfaction of a formula in the first position of a model, then, as there is no past from the first position, a formula with past can be converted to a pure future formula [22, 26]. The conversion, however, can be exponential [34].

Formally, for a formula φ and a position $i \geq 0$, we say that φ *holds at position* i *of* σ, written $\sigma, i \models \varphi$, and define it inductively as follows:

- For $p \in P$ we have $\sigma, i \models p$ iff $p \in \sigma_i$.
- $\sigma, i \models \neg\varphi$ iff $\sigma, i \not\models \varphi$.
- $\sigma, i \models \varphi \vee \psi$ iff $\sigma, i \models \varphi$ or $\sigma, i \models \psi$.
- $\sigma, i \models \bigcirc \varphi$ iff $\sigma, i+1 \models \varphi$.
- $\sigma, i \models \varphi \mathscr{U} \psi$ iff there exists $k \geq i$ such that $\sigma, k \models \psi$ and $\sigma, j \models \varphi$ for all j, $i \leq j < k$.
- $\sigma, i \models \ominus \varphi$ iff $i > 0$ and $\sigma, i-1 \models \varphi$.
- $\sigma, i \models \varphi \mathscr{S} \psi$ iff there exists $k, 0 \leq k \leq i$ such that $\sigma, k \models \psi$ and $\sigma, j \models \varphi$ for all j, $k < j \leq i$.

We note that the interpretation of future and past is non-strict. The until operator interpreted in location i characterizes the locations greater than i as well as i itself; and similarly for the past. If $\sigma, 0 \models \varphi$, then we say that φ *holds* on σ and denote

it by $\sigma \models \varphi$. A set of models M satisfies φ, denoted $M \models \varphi$, if every model in M satisfies φ.

We use the usual abbreviations of the Boolean connectives $\wedge$, $\rightarrow$ and $\leftrightarrow$ and the usual definitions for true and false. We introduce the following temporal abbreviations $\diamondsuit$ (eventually), $\square$ (globally), $\mathscr{W}$ (weak-until), and for the past fragment $\boxminus$ (historically), $\diamondminus$ (once), and $\mathscr{B}$ (back-to) which are defined as follows.

- $\diamondsuit \phi \equiv \mathsf{true}\, \mathscr{U}\, \phi$,
- $\square \psi \equiv \neg \diamondsuit \neg \psi$,
- $\varphi \mathscr{W} \psi \equiv (\varphi \mathscr{U} \psi) \vee \square \varphi$,
- $\diamondminus \phi \equiv \mathsf{true}\, \mathscr{S}\, \phi$,
- $\boxminus \psi \equiv \neg \diamondminus \neg \varphi$, and
- $\varphi \mathscr{B} \psi \equiv (\varphi \mathscr{S} \psi) \vee \boxminus \varphi$.

For example, the formula $\varphi_1 \equiv \square(p \rightarrow \diamondsuit q)$ holds in models in which every location where p is true is followed later (or concurrently) by a location where q holds. A location where p happened in the past but no q happened since satisfies $\neg q \mathscr{S} (\neg q \wedge p)$. Thus, $\varphi_2 \equiv \square \diamondsuit (\neg(\neg q \mathscr{S} (\neg q \wedge p)))$ means that there is no location where p holds and no q occurs after it, i.e., when checked in the beginning of a model it is the same as $\square(p \rightarrow \diamondsuit q)$.

Another common notation for the temporal operators is **X** for next, **Y** for previous, **F** for eventually (or future), **G** for globally, **H** for historically, and **P** for once (past). Using this notation $\square(p \rightarrow \diamondsuit q)$ becomes $\mathbf{G}(p \rightarrow \mathbf{F}q)$ and $\square(p \rightarrow \bigcirc q)$ becomes $\mathbf{G}(p \rightarrow \mathbf{X}q)$.

For an LTL formula φ, we denote by $\mathscr{L}(\varphi)$ the set of models that satisfies φ. That is,

$$\mathscr{L}(\varphi) = \{\sigma \mid \sigma \models \varphi\}.$$

We are mostly interested in when an LTL formula is satisfied in the first location in a sequence. However, generally, the satisfaction is related to a location. This duality gives rise to two notions of equivalence for LTL formulas. We say that two LTL formulas φ and ψ are *equivalent* if $\mathscr{L}(\varphi) = \mathscr{L}(\psi)$. We say that two LTL formulas φ and ψ are *globally equivalent* if for every model σ and every location $i \geq 0$ we have $\sigma, i \models \varphi$ iff $\sigma, i \models \psi$. For example, the two formulas mentioned above, φ_1 and φ_2, are equivalent but not globally equivalent. Indeed, consider a model where p holds in the first location and both p and q are false forever after that. In the second location in this sequence the formula φ_1 holds. However, the formula φ_2 does not hold in the second location as it notices the p in the first location is never followed by a q.

2.3.2 Safety Versus Liveness and the Temporal Hierarchy

A very important distinction in LTL formulas is between safety and liveness. Intuitively, a safety property says that bad things will never happen. A liveness property

says that the system will also do good things. Algorithmically, safety is much easier to check than liveness and this is the most prevalent form of specification that is encountered in practice. Nevertheless, we further refine the properties expressible in LTL to a *temporal hierarchy* that relates to expressiveness, the topology of sets of models characterized by them, and the types of deterministic FDSs (or automata) that accept them. Formally, we have the following.

A property φ is a *safety* property for a set of models if for every model σ that violates φ, i.e., $\sigma \not\models \varphi$, there exists an i such that for every σ' that agrees with σ up to position i, i.e., $\forall 0 \leq j \leq i, \sigma'_i = \sigma_i$, σ' also violates φ. As mentioned, safety properties specify bad things that should never happen. Thus, once an error has occurred (in location i), it is impossible to undo it. Every possible extension also includes the same error and is *unsafe*.

A property φ is a *liveness* property for a set of models if for every prefix of a model $w_0, \ldots, w_i$ there exists an infinite model σ that starts with $w_0, \ldots, w_i$ and $\sigma \models \varphi$. As mentioned liveness properties specify good things that should occur. Thus, regardless of the history of the computation, it is still possible to find an extension that will fulfill the specification.

One prevalent form of safety is the *invariant*, a formula of the form $\Box p$ where p is propositional. In general, safety properties are easier to check than liveness properties. As we are interested in violations of safety, we may restrict attention to finite paths and hence to reachability of violations. In particular, for every safety property φ and an FDS $\mathscr{D}$, it is possible to construct an FDS $\mathscr{D}_1$ and an invariant $\Box p$ such that there is no initial path in $\mathscr{D}$ that violates φ iff all reachable states of $\mathscr{D}_1$ satisfy p. Checking p over $\mathscr{D}_1$ is much easier to do and this is indeed the way safety properties are checked in practice. Compare this with the more complex algorithm for viability in the presence of justice in Sect. 2.2 (in the next subsection we see that this is required for LTL model checking even if the model is fairness free).

More theoretically, Alpern and Schneider [1] show that every language can be described as the intersection of a liveness and a safety property (not restricted to properties expressed in LTL).

Theorem 2 *Every language $L \subseteq (2^P)^\omega$ is expressible as the intersection $L_s \cap L_l$, where L_s is a safety property and L_l is a liveness property.*

Proof Consider a language $L \subseteq (2^P)^\omega$. We have the following definitions.

$$\begin{aligned} \mathit{pref}(L) &= \{w_0, \ldots, w_n \mid w_0, \ldots, w_n \text{ is a prefix of some } w \in L\}, \\ L_s &= \left\{w_0, w_1, \ldots \,\middle|\, \text{for every } i,\ w_0, \ldots, w_i \in \mathit{pref}(L)\right\}, \\ L_l &= L \cup \left\{w_0, w_1, \ldots \,\middle|\, \text{there exists } i \text{ such that } w_0, \ldots, w_i \notin \mathit{pref}(L)\right\}. \end{aligned}$$

It is simple to see that L_s is a safety property. In order to see that L_l is a liveness property consider some prefix $w_0, \ldots, w_i$. If $w_0, \ldots, w_i \in \mathit{pref}(L)$, then clearly there is some extension σ of $w_0, \ldots, w_i$ such that $\sigma \in L \subseteq L_l$. Otherwise, if $w_0, \ldots, w_i \notin \mathit{pref}(L)$ then all extensions σ of $w_0, \ldots, w_i$ are in L_l.

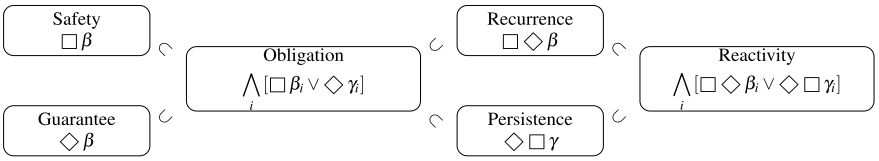


Fig. 4 The temporal hierarchy, where β, γ, β_i, and γ_i are pure past temporal properties

Finally, it is simple to see that $L \subseteq L_s$ and $L \subseteq L_l$ hence $L \subseteq L_s \cap L_l$. In the other direction, consider a word $\sigma \in L_s \cap L_l$. It follows that either $\sigma \in L_s \cap L$ or $\sigma \in L_s \cap \{w_0, w_1, \ldots \mid \text{there is } i \text{ such that } w_0, \ldots, w_i \notin \mathit{pref}(L)\}$. In the first case, clearly $\sigma \in L$. The second intersection must be empty as σ cannot at the same time have all its prefixes in $\mathit{pref}(L)$ and some prefix not in $\mathit{pref}(L)$. □

We now discuss a more refined classification of temporal properties. Although Safety, as appearing in this classification, seems different from the description of safety above, Theorem 3 shows that they are actually the same. This classification is important, for example and as described later in this Handbook, in synthesis ([6] (Bloem et al., Graph Games and Reactive Synthesis)) and deductive verification ([48] (Shankar, Combining Model Checking and Deduction)). In Fig. 4 we define six different classes of properties. In both synthesis and deductive verification, formulas in one of these classes have a better (i.e., more efficient or simpler) treatment than formulas in higher classes. Furthermore, this classification is related to the ability to convert such properties to deterministic ω-automata, see later in this Handbook [30] (Kupferman, Automata Theory and Model Checking), and to the topological complexity of properties.

We do not state formally most properties of this hierarchy. As shown in Fig. 4, all containments are strict. Furthermore, there are Safety properties that are not Guarantee properties and vice versa. Similarly, there are Recurrence properties that are not Persistence properties and vice versa. Each of Obligation and Reactivity form a strict hierarchy according to the number of conjuncts. The lowest in the Obligation and Reactivity hierarchies (i.e., one conjunct) contain all the classes below them. Then, all classes are closed under disjunction and conjunction and Obligation and Reactivity are also closed under negation. The complement of every Safety property is a Guarantee property and vice versa. Similarly, the complement of every Recurrence property is a Persistence property and vice versa.

We state formally that the classification is exhaustive.

Theorem 3 *Every Safety property expressible in LTL can be expressed as a formula of the form* $\Box\beta$, *where* β *is a pure past formula. Every LTL formula can be expressed as a Reactivity formula.*

We note that the translation to this normal form causes an explosion in the size of the formula. The translation uses as subroutines translations between LTL, automata, and regular expressions [22]. A characterization of safety and liveness in

terms of pure-future LTL and decision procedures for whether a formula is a safety or liveness property are given in [49].

Further details about these classes, formal statement of the claims above, their proofs, and the relation of these classes to topology and to automata on infinite words are available in [39]. The intersection of Safety and Guarantee is studied in [32]. Exhaustiveness of the hierarchy is covered in [37].

2.3.3 Extensions of LTL

For many years logicians have been studying the first-order logic and second-order logic of infinite sequences, denoted FOL1 and S1S, respectively. In our context, these logics are restricted to use the relations $<$ and $=$, and the function $+1$. Naturally, LTL was compared with these logics. It was shown that LTL and FOL1 are equally expressive [26]. That is, for every FOL1 formula there is an LTL formula that characterizes the same models. As the semantics of LTL is expressed in FOL1, it is quite clear that it cannot be more expressive that FOL1. Showing that LTL is as expressive as FOL1 is more complicated and is not covered here. This observation led to the declaration that "LTL is expressively complete" [22]. However, very natural properties that are required for specifying programs cannot be expressed in LTL [55]. Following this realization different studies of how to extend the expressive power of LTL so that it becomes equivalent to S1S followed, culminating in the definition of PSL (see [15] in this Handbook).

S1S formulas extend Boolean connectives by using location and predicate variables. We assume a countable set of location variables $var = \{x, y, \ldots\}$ and a countable set of predicate variables $VAR = \{X, Y, \ldots\}$. We define the set of terms (τ), atomic formulas (α), and formulas (φ) of S1S.

$$\begin{array}{lll} \tau & ::= & x \mid \tau + 1, \\ \alpha & ::= & X(\tau) \mid p(\tau) \mid \tau_1 < \tau_2 \mid \tau_1 = \tau_2, \\ \varphi & ::= & \alpha \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \exists x.\varphi \mid \exists X.\varphi. \end{array} \tag{2}$$

FOL1 is obtained by allowing only the use of location variables.

We evaluate the truth of formulas over models (as before) augmented with valuations $v : var \to \mathbb{N}$ and $V : VAR \to 2^{\mathbb{N}}$. Given a valuation v we denote by $v[x \mapsto i]$ the valuation that assigns $v(y)$ to every $y \neq x$ and assigns i to x. Similarly, we denote by $V[X \mapsto I]$ the valuation that assigns $V(Y)$ for every $Y \neq X$ and assigns I to X. For uniformity of notation, given a model $\sigma = w_0, w_1, \ldots$ we treat a proposition p as a subset of $\mathbb{N}$, where $i \in p$ iff $p \in w_i$. Given a model $\sigma = w_0, w_1, \ldots$ over a set of propositions P, and valuations v and V the semantics of an S1S formula is defined as follows (Boolean connectives omitted).

- For a term τ, we define $v(\tau)$ as expected: for a variable x, $v(x)$ is given by the valuation v and $v(\tau + 1) = v(\tau) + 1$.
- $\sigma, v, V \models X(\tau)$ iff $v(\tau) \in V(X)$.

- $\sigma, v, V \models p(\tau)$ iff $v(\tau) \in p$.
- $\sigma, v, V \models \tau_1 < \tau_2$ iff $v(\tau_1) < v(\tau_2)$.
- $\sigma, v, V \models \tau_1 = \tau_2$ iff $v(\tau_1) = v(\tau_2)$.
- $\sigma, v, V \models \exists x.\varphi$ iff there is some $i \in \mathbb{N}$ such that $\sigma, v[x \mapsto i], V \models \varphi$.
- $\sigma, v, V \models \exists X.\varphi$ iff there is some $I \subseteq \mathbb{N}$ such that $\sigma, v, V[X \mapsto I] \models \varphi$.

If all variables in φ are bound by quantifiers, then the initial valuations v and V are not important and we may write $\sigma \models \varphi$. We introduce the shorthands $\forall x.\varphi \equiv \neg\exists x.\neg\varphi$, $\forall X.\varphi \equiv \neg\exists X.\neg\varphi$, and the formula $\tau_1 \leq \tau_2 \equiv \tau_1 < \tau_2 \vee \tau_1 = \tau_2$. Finally, using the formula $\exists z.\forall y.z \leq y$ we can easily express the term 0. We can now show that LTL is less expressive than S1S.

Lemma 6 *The property "p holds only in even positions" is expressible in S1S but not in LTL.*

Proof The following S1S formula expresses this property.

$$\exists T.\big(T(0) \wedge \big(\forall x.T(x) \leftrightarrow \neg T(x+1)\big) \wedge \big(\forall x.p(x) \rightarrow T(x)\big)\big).$$

That is, T must be assigned the set of even numbers, indeed, T must include 0 and exactly one of every two consecutive numbers. Then, whenever p is true, it must be the case that T is true as well, implying that the location is even.

We show that this cannot be expressed in LTL. Here we consider LTL restricted to future operators. We omit the longer proof (of a very similar flavor).

Consider the family of models $r_i = w_0^i, w_1^i, \ldots$ over the proposition p, where $w_i^i = \{p\}$ and $w_j^i = \emptyset$ for $j \neq i$. That is, in the i-th model, the proposition p is true exactly at the i-th location and nowhere else. We show that for every future LTL formula φ there is some n large enough so that φ cannot distinguish r_n from r_{n+1}.

For a future LTL formula φ, let $n(\varphi)$ denote the number of next operators used in φ. We show by induction on the structure of the formula that for every $i > n(\varphi)$ we have $r_i \models \varphi$ iff $r_{i+1} \models \varphi$. For propositions, Boolean operators, and formulas of the form $\bigcirc\psi$ the proof is simple.

Consider the case that $\varphi = \psi_1 \mathscr{U} \psi_2$. By assumption, for ψ_k, $k = 1, 2$, and for every $i > n(\psi_k)$ we have $r_i \models \psi_k \leftrightarrow r_{i+1} \models \psi_k$. Clearly, $n(\varphi) \geq \max(n(\psi_1), n(\psi_2))$. Consider some $i > n(\varphi)$. Suppose that $r_i \models \varphi$, then there is some $k \geq 0$ such that $r_i, k \models \psi_2$ and for every $0 \leq j < k$ we have $r_i, j \models \psi_1$. If $k = 0$, then by assumption $r_i \models \psi_2$ iff $r_{i+1} \models \psi_2$ and we are done. Otherwise, the first suffix of r_{i+1} is r_i and hence $r_{i+1}, 1 \models \varphi$. However, r_i and r_{i+1} agree on the satisfaction of ψ_1 implying that $r_{i+1}, 0 \models \psi_1$. Thus, $r_{i+1} \models \varphi$. As $\neg\varphi$ is $(\neg\psi_2) \mathscr{W} (\neg\psi_1 \wedge \neg\psi_2)$ the proof in the case that φ does not hold is similar. □

Corollary 1 *LTL is less expressive than S1S.*

The question now arises, what needs to be added to LTL in order to increase its expressive power. The original solution suggested in [55] was to add left-linear grammars, i.e., regular languages. This was later revised and extended by using

finite automata connectives [54]. An alternative solution was to add to LTL quantification over predicates [51]. The resulting logic, QPTL, has a similar flavor to S1S.

Here, in order to increase the expressive power of LTL we are going to add automata connectives to it. The resulting logic, called ETL, is defined as follows. As before, we consider a countable set of propositions.

$$\varphi ::= p \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid A(\varphi_1, \ldots, \varphi_n). \tag{3}$$

That is, an automaton A over alphabet $\{a_1, \ldots, a_n\}$ can be used as an n-ary operator $A(\psi_{a_1}, \ldots, \psi_{a_n})$.

For an automaton operator $\varphi = A(\psi_{a_1}, \ldots, \psi_{a_n})$ we define the satisfaction relation $\sigma, i \models \varphi$ as follows.

- $\sigma, i \models A(\psi_{a_1}, \ldots, \psi_{a_n})$ iff there is a word $b_0 b_1 \cdots b_{m-1}$ accepted by A and for every $0 \leq j < m$ we have $\sigma, i + j \models \psi_{b_j}$.

It is simple enough to see that future LTL can be easily expressed in ETL. Indeed, $\bigcirc \psi$ is $A(\mathsf{true}, \psi)$, where A is an automaton with alphabet $\{a_1, a_2\}$ accepting exactly the word $a_1 a_2$. Similarly, $\psi_1 \mathcal{U} \psi_2$ is $A(\psi_1, \psi_2)$, where A is an automaton with alphabet $\{a_1, a_2\}$ accepting the regular expression $a_1^* a_2$. Also, the problematic property "p holds only in even positions" can be expressed by $\neg A(\mathsf{true}, p)$, where A is the automaton accepting $a_1(a_1 a_1)^* a_2$. That is, it cannot be the case that the distance to a location where p holds is an odd number. The proof that ETL is as expressive as S1S is quite complex. It uses the proof that S1S is equally expressive as nondeterministic Büchi automata (see elsewhere in this Handbook in [30] (Kupferman, Automata Theory and Model Checking)) and a proof that the language of a nondeterministic Büchi automaton can be expressed in ETL. The interested reader is referred to [54].

Theorem 4 *ETL and S1S are equally expressive.*

We now turn our attention to a different extension of LTL. So far, we have restricted our attention to atomic formulas that are propositions, i.e., Boolean variables. However, the FDSs defined in Sect. 2.2 were more general; we allowed variables ranging over other discrete domains. In order to reason about such FDSs, we introduce first-order elements into LTL. In general, incorporating a first-order part into LTL results in a very expressive language. Here, we concentrate on a fragment that is expressive enough to reason about FDS. This will be particularly useful later in this Handbook in [25] and [3] (Barrett, Satisfiability Modulo Theories). Even more general definitions of FOLTL are described, e.g., in [16, 29].

In order to reason about general FDS we consider models with more general letters. Namely, instead of letters that are truth assignments to propositions we consider letters that are first-order models. We generalize the set of propositions to a set of n-ary predicates for $n \geq 0$, where propositions are 0-ary predicates. We denote by R the set of predicates and use $r, s, \ldots$ to range over individual predicates. We

use the symbols $f, g, \ldots$ to denote n-ary functions for $n \geq 1$ and $c, d, \ldots$ to denote constants (or 0-ary functions). For example, $*$ and $+$ are now two binary functions with the expected meanings. Let $var = \{x, y, \ldots\}$ be a countable set of variables. We define the set of terms (τ), atomic formulas (α), and formulas (φ) of FOLTL.

$$\begin{array}{l} \tau \ ::= \ c \mid x \mid \bigcirc x \mid f(\tau, \ldots, \tau), \\ \alpha \ ::= \ r(\tau_1, \ldots, \tau_n) \mid \neg\alpha \mid \alpha_1 \vee \alpha_2 \mid \exists x.\alpha, \\ \varphi \ ::= \ \alpha \mid \varphi_1 \vee \varphi_2 \mid \neg\varphi \mid \bigcirc\varphi \mid \ominus\varphi \mid \varphi_1 \mathcal{U} \varphi_2 \mid \varphi_1 \mathcal{S} \varphi_2. \end{array} \tag{4}$$

A model for FOLTL is $\sigma : w_0, w_1, \ldots$, where w_i gives an interpretation for the predicate, function, constant, and variable symbols used in the formula. Given a model $\sigma = w_0, w_1, \ldots$ over propositions P, variables var, functions F, and predicates R, the semantics of an FOLTL formula is defined as follows (for cases that were not previously defined).

- For a term τ and a letter w_i we define $w_i(\tau)$ as expected; for a variable x, $w_i(x)$ is the interpretation of x in w_i, $w_i(\bigcirc x)$ is $w_{i+1}(x)$, $w_i(c)$ is the interpretation of c in w_i, and $w_i(f(\tau_1, \ldots, \tau_n))$ is $w_i(f)(w_i(\tau_1), \ldots, w_i(\tau_n))$.
- For an atomic formula α and a model σ, we define when σ satisfies α in location i, denoted $\sigma, i \models \alpha$, as follows.
  - $\sigma, i \models r(\tau_1, \ldots, \tau_n)$ iff $w_i(r)(w_i(\tau_1), \ldots, w_i(\tau_n)) = \mathsf{true}$.
  - $\sigma, i \models \neg\alpha$ iff $\sigma, i \not\models \alpha$.
  - $\sigma, i \models \alpha_1 \vee \alpha_2$ iff $\sigma, i \models \alpha_1$ or $\sigma, i \models \alpha_2$.
  - $\sigma, i \models \exists x.\alpha$ iff there is a model $\tilde{\sigma}$ such that $\tilde{\sigma}$ agrees with σ on all locations different from i and $\tilde{w}_i$ agrees with w_i on the interpretation of all functions, predicates, constants, and variables different from x such that $\tilde{\sigma}, i \models \alpha$.

Notice that the relations $<$ or $=$ can now be defined as binary predicates. This definition is quite general. In particular, it allows every letter in the model to interpret a predicate in a different way. For the purpose of this book it is mostly enough to assume that predicates will be interpreted in the natural way and will not change their meaning between different states. Notice that quantification is local to one location, this can be made more general.

Given this definition of FOLTL, for every FDS $\mathscr{D}$, there exists an FOLTL formula $\varphi_{\mathscr{D}}$, called the *temporal semantics* of $\mathscr{D}$, which characterizes the computations of $\mathscr{D}$. It is given by:

$$\varphi_{\mathscr{D}}: \quad \theta \wedge \Box(\rho(\mathscr{V}, \bigcirc\mathscr{V})) \wedge \bigwedge_{J \in \mathscr{J}} \Box\Diamond J \wedge \bigwedge_{(P,Q) \in \mathscr{C}} (\Box\Diamond P \rightarrow \Box\Diamond Q),$$

where $\rho(\mathscr{V}, \bigcirc\mathscr{V})$ is the formula obtained from ρ by replacing each instance of the primed variable v' by the LTL formula $\bigcirc v$. Note that we assume some fixed definition of relations (such as $<$ and $=$) and functions (such as $+$, $+1$, or $*$) that may be used in the definition of $\mathscr{D}$.

2.3.4 Temporal Testers, Satisfiability, and Model Checking

Given an LTL formula we convert it to an FDS that recognizes its truth or falsity. We start with construction of a temporal tester, an FDS that merely monitors the truth value of the formula in a computation. That is, the temporal tester has a computation for every possible sequence of truth assignments to propositions. It then annotates this sequence of truth assignments with the value of the formula in every location. A temporal tester can be thought of as adding a new proposition that marks the truth value of the formula. We then use these FDSs to check the satisfiability of a formula or for model checking.

Definition 5 (Temporal tester) A *temporal tester* for a formula φ is a JDS $\mathscr{T}_\varphi$ that has a distinguished Boolean variable x_φ such that the following hold. Let P be the set of propositions appearing in φ.

For every computation $\sigma : s_0, s_1, \ldots$ of $\mathscr{T}_\varphi$ we have $s_i[x_\varphi] = 1$ iff $(\sigma, i) \models \varphi$.

For every sequence of states $\pi : t_0, t_1, \ldots$ in $(\Sigma_P)^\omega$ there is a computation $\sigma : s_0, s_1, \ldots$ of $\mathscr{T}_\varphi$ such that for every i we have $s_i \Downarrow_P = t_i$.

We show how to construct testers by induction on the structure of the formula. Thus, we construct a small FDS for each formula ψ using Boolean variables that signal the truth values of the subformulas of ψ. We then take the synchronous parallel composition of these FDS for all the subformulas of φ.

- For a proposition p, we have $T_p = \langle \{p\}, \mathsf{true}, \mathsf{true} \rangle$.
- For a formula $\psi = \neg\psi_1$, we have $T_\psi = \langle \{x_\psi, x_{\psi_1}\}, \mathsf{true}, x_\psi = \neg x_{\psi_1} \rangle$.
- For a formula $\psi = \psi_1 \vee \psi_2$, we have $T_\psi = \langle \{x_\psi, x_{\psi_1}, x_{\psi_2}\}, \mathsf{true}, (x_\psi = (x_{\psi_1} \vee x_{\psi_2})) \rangle$.
- For a formula $\psi = \bigcirc \psi_1$, we have $T_\psi = \langle \{x_\psi, x_{\psi_1}\}, \mathsf{true}, (x_\psi = x'_{\psi_1}) \rangle$.
- For a formula $\psi = \psi_1 \mathscr{U} \psi_2$, we have

$$T_\psi = \left\langle \{x_\psi, x_{\psi_1}, x_{\psi_2}\}, \mathsf{true}, x_\psi = \left(x_{\psi_2} \vee \left(x_{\psi_1} \wedge x'_\psi\right)\right), \{\neg x_\psi \vee x_{\psi_2}\} \right\rangle.$$

- For a formula $\psi = \ominus \psi_1$, we have $T_\psi = \langle \{x_\psi, x_{\psi_1}\}, \neg x_\psi, x'_\psi = x_{\psi_1} \rangle$.
- For a formula $\psi = \psi_1 \mathscr{S} \psi_2$, we have

$$T_\psi = \left\langle \{x_\psi, x_{\psi_1}, x_{\psi_2}\}, x_\psi = x_{\psi_2}, x'_\psi = \left(x'_{\psi_2} \vee \left(x'_{\psi_1} \wedge x_\psi\right)\right) \right\rangle.$$

Notice that the only case that the temporal tester requires fairness (justice) is the case of until. Also the temporal testers for past operators are set to a definite value. This is due to the knowledge of the (nonexistent) past. For an LTL formula φ let sub(φ) denote the set of subformulas of φ, according to the grammar in Eq. (1). Consider an LTL formula φ, let sub(φ) $= \{\psi_1, \ldots, \psi_n\}$. Then, the temporal tester for φ is $\mathscr{T}_\varphi = T_{\psi_1} ||| \cdots ||| T_{\psi_n}$.

Theorem 5 *The JDS $\mathscr{T}_\varphi$ is a temporal tester for φ.*

While a tester signals the truth value of every subformula in every location in a computation, we are sometimes interested in an FDS all of whose computations satisfy the formula. For that, we specialize the tester into an acceptor.

Definition 6 (Acceptor) An *acceptor* for a formula φ is a JDS $\mathscr{A}_\varphi$ such that the following hold. Let P be the set of propositions appearing in φ.

For every computation $\sigma : s_0, s_1, \ldots$ of $\mathscr{A}_\varphi$ we have $\sigma \models \varphi$.

For every model $\pi : t_0, t_1, \ldots$ in $(\Sigma_P)^\omega$ such that $\pi \models \varphi$ there is a computation $\sigma : s_0, s_1, \ldots$ of $\mathscr{A}_\varphi$ such that for every i we have $s_i \Downarrow_P = t_i$.

An acceptor is a mild variant of a temporal tester. Indeed, all we have to do is to add an initial condition demanding that the variable x_φ is true in the initial state. Formally, for a formula φ let $A_\varphi = \langle \{x_\varphi\}, x_\varphi, \mathsf{true} \rangle$. Then, $\mathscr{A}_\varphi = A_\varphi \,|||\, \mathscr{T}_\varphi$.

Theorem 6 *The JDS $\mathscr{A}_\varphi$ is an acceptor for φ.*

For further details on temporal testers and proofs of Theorems 5 and 6 we refer the reader to [43].

We are finally ready to define the notion of linear-time model checking. While satisfiability calls for finding a model that satisfies a logical formula, model checking in fact seeks the validity of a formula over the set of models produced by an FDS. Algorithmically, as we show below, we check validity by reducing it to non-satisfiability of the complement. So both validity and model checking are reduced to satisfiability, the first by considering the satisfiability of the complement, the second by considering the satisfiability of the complement over the FDS representing the system.

Definition 7 (Satisfiability) An LTL formula φ is satisfiable if its set of models is non-empty. That is, φ is satisfiable if $\mathscr{L}(\varphi) \neq \emptyset$.

Theorem 7 *LTL satisfiability is decidable in polynomial space.*

Proof Consider an acceptor $\mathscr{A}_\varphi$ for φ. Clearly, if $\mathsf{Comp}(\mathscr{A}_\varphi) \neq \emptyset$ we can conclude that φ has some model. As $\mathscr{A}_\varphi$ is a JDS, Algorithm 3 can check whether $\mathscr{A}_\varphi$ has some computation. The number of Boolean variables used in the acceptor $\mathscr{A}_\varphi$ is proportional to the size of the formula φ. Thus, in the worst case, the number of states of $\mathscr{A}_\varphi$ is exponential in the size of φ and the polynomial space upper bound follows from the logarithmic space algorithm for checking viability. □

Definition 8 (Validity) An LTL formula φ is valid if every sequence in $(\Sigma_{P'})^\omega$ is a model of φ, where $P_\varphi \subseteq P'$ is the set of propositions appearing in φ.

Theorem 8 *LTL validity is decidable in polynomial space.*

Theorem 8 follows from the fact that an LTL formula φ is valid iff $\neg\varphi$ is not satisfiable.

Definition 9 (Implementation) We say that an FDS $\mathscr{D}$ *implements* specification φ, denoted $\mathscr{D} \models \varphi$, if every computation of $\mathscr{D}$ satisfies φ.

Definition 10 (Model checking) Given an FDS $\mathscr{D}$ and an LTL formula φ, the model-checking problem for $\mathscr{D}$ and φ is to decide whether $\mathscr{D}$ implements φ.

Theorem 9 *LTL model checking is decidable in polynomial space.*

In the proof below, the model-checking problem for an FDS $\mathscr{D}$ and an LTL formula φ is solved by trying to find a single computation of $\mathscr{D}$ that does not satisfy φ. For that, we use the acceptor for $\neg\varphi$.

Proof Consider the acceptor $\mathscr{A}_{\neg\varphi}$ and the synchronous composition $\mathscr{K} = \mathscr{A}_{\neg\varphi} ||| \mathscr{D}$. Suppose that $\mathscr{K}$ is viable. That is, there is some computation σ of $\mathscr{K}$. By Lemma 2 the projection of σ on the variables of $\mathscr{D}$ is a computation of $\mathscr{D}$ and the projection of σ on the variables of $\mathscr{A}_{\neg\varphi}$ is a computation of $\mathscr{A}_{\neg\varphi}$. By Theorem 6, a computation of $\mathscr{A}_{\neg\varphi}$ is a model of $\neg\varphi$. It follows that σ is a computation of $\mathscr{D}$ that does not satisfy φ.

In the other direction, suppose that σ is a computation of $\mathscr{D}$ that does not satisfy φ. Then, by Theorem 6, there is a computation σ' of $\mathscr{A}_{\neg\varphi}$ that agrees with σ on all the propositions. It follows that $\sigma ||| \sigma'$, where in every position $\sigma ||| \sigma'$ agrees with σ on the variables of $\mathscr{D}$ and with σ' on the variables of $\mathscr{A}_{\neg\varphi}$, is a computation of $\mathscr{D} ||| \mathscr{A}_{\neg\varphi}$. □

We note that the complexity results for satisfiability and model checking were obtained initially with different techniques from those described here [50].

We now turn our attention to a different approach to specifying properties of programs.

2.4 Computation Tree Logic

In this section we present branching-time logic and computation tree logic (abbreviated CTL). In the branching-time view there may be multiple possible futures in a given state, corresponding to its different successors. A computation, then, is a single structure that captures the branching of the entire program. Originally, CTL would view a program as a generator of a computation tree—a tree structure in which every node is labeled by states of the program and the same state may label different nodes on the same branch of the tree (corresponding to a loop in the program). Here, we choose to define CTL satisfaction directly over FDSs and not over their computation trees. Thus, a CTL formula characterizes a set of possible systems. We define CTL and study some extensions of it. We then connect this back to model checking and show how to solve the model checking problem for CTL.

2.4.1 Definition of Computation Tree Logic

We assume a countable set of Boolean propositions P. Here, we choose to define fairness-free FDSs as the models of CTL formulas. We view an FDS as giving truth assignments to propositions by demanding that the propositions are Boolean variables of the FDS. Namely, if $\hat{P}$ is the set of propositions appearing in φ, then every FDS $\mathscr{D}$ such that $\hat{P} \subseteq \mathscr{V}$ is a model for φ.

CTL formulas use the Boolean connectives and augment the temporal connectives from LTL with path quantification. Each temporal connective is combined with either "for all" or "for some" paths, giving rise to interpretation over the states of the FDS.

$$\varphi ::= p \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \mathbf{A}\bigcirc(\varphi) \mid \mathbf{A}(\varphi_1 \,\mathscr{U}\, \varphi_2) \mid \mathbf{A}(\varphi_1 \,\mathscr{W}\, \varphi_2). \tag{5}$$

The intuitive meaning of **A** is that the temporal formula nested within should hold for all paths that start in a state.

Formally, for a formula φ and an FDS $\mathscr{D} = \langle \mathscr{V}, \theta, \rho \rangle$, we say that φ *holds in state s of* $\mathscr{D}$, written $\mathscr{D}, s \models \varphi$, and define it inductively as follows:

- For $p \in P$ we have $\mathscr{D}, s \models p$ iff $s \models p$.
- $\mathscr{D}, s \models \neg\varphi$ iff $\mathscr{D}, s \not\models \varphi$.
- $\mathscr{D}, s \models \varphi_1 \vee \varphi_2$ iff $\mathscr{D}, s \models \varphi_1$ or $\mathscr{D}, s \models \varphi_2$.
- $\mathscr{D}, s \models \mathbf{A}\bigcirc(\varphi)$ if for every t such that $(s, t') \models \rho$ we have $\mathscr{D}, t \models \varphi$.
- $\mathscr{D}, s \models \mathbf{A}(\varphi_1 \,\mathscr{U}\, \varphi_2)$ if for every path $\sigma = s_0, s_1, \ldots$ starting in s there is some $0 \leq i < |\sigma|$ such that $\mathscr{D}, s_i \models \varphi_2$ and for every $0 \leq j < i$ we have $\mathscr{D}, s_j \models \varphi_1$.
- $\mathscr{D}, s \models \mathbf{A}(\varphi_1 \,\mathscr{W}\, \varphi_2)$ if for every path $\sigma = s_0, s_1, \ldots$ starting in s and for every $0 \leq j < |\sigma|$ either $\mathscr{D}, s_j \models \varphi_1$ or there is $0 \leq i \leq j$ such that $\mathscr{D}, s_i \models \varphi_2$.

We use the usual abbreviations of the Boolean connectives and the temporal connectives $\diamondsuit$ and $\square$ as in LTL. We introduce the path quantifier **E** as an abbreviation, as follows.

- $\mathbf{E}\bigcirc(\varphi) \equiv \neg\mathbf{A}\bigcirc(\neg\varphi)$,
- $\mathbf{E}(\varphi_1 \,\mathscr{U}\, \varphi_2) \equiv \neg\mathbf{A}(\neg\varphi_2 \,\mathscr{W}\, (\neg\varphi_1 \wedge \neg\varphi_2))$,
- $\mathbf{E}(\varphi_1 \,\mathscr{W}\, \varphi_2) \equiv \neg\mathbf{A}(\neg\varphi_2 \,\mathscr{U}\, (\neg\varphi_1 \wedge \neg\varphi_2))$.

For example, the formula $\mathbf{E}\bigcirc(p)$ holds in a state that has some successor in which proposition p holds. The formula $\mathbf{E}(p \,\mathscr{U}\, q)$ holds in a state from which there is a path over which $p \,\mathscr{U}\, q$ holds. We note that FDSs are assumed to be fairness-free and, in general, CTL does not use the concept of runs and computations.

Also here, sometimes letters replace the symbols and $\mathbf{A}\bigcirc$ becomes **AX**, $\mathbf{A}\diamondsuit$ becomes **AF**, and $\mathbf{A}\square$ becomes **AG**, and similarly for the existential path quantification.

For a CTL formula φ and an FDS $\mathscr{D}$, we denote by $[\![\mathscr{D}, \varphi]\!]$ the set of states for which $\mathscr{D}, s \models \varphi$. Formally, $[\![\mathscr{D}, \varphi]\!] = \{s \in \Sigma_{\mathscr{V}} \mid \mathscr{D}, s \models \varphi\}$.

Definition 11 (Implementation) We say that an FDS $\mathscr{D}$ *implements* specification φ, denoted $\mathscr{D} \models \varphi$, if every initial state of $\mathscr{D}$ satisfies the formula.

2.4.2 Extensions

We consider extensions of CTL that are similar to those considered for LTL. Due to the branching nature of CTL, the definition of past is more involved and we treat it here. We then proceed to consider FOCTL. These are presented more briefly than the case for LTL as they are used less often in practical model checking.

When coming to extend CTL with past we have to make several decisions. Firstly, we need to decide whether the past is branching or linear. In the first interpretation, every state has all its predecessors as possible pasts. In the second interpretation we consider an unwinding of the system to a tree (from a certain state) and every node in this tree has a unique past. Secondly, we need to decide whether the past may be infinite. Here, we choose to define a branching but finite past. This decision somewhat simplifies the treatment of CTL with past. For an in-depth treatment of all options we refer the reader to [31].

Past CTL formulas augment the definition of CTL by including the following clauses in the grammar in Eq. (5).

$$\varphi ::= \mathbf{A} \ominus(\varphi) \mid \mathbf{A}(\varphi_1 \mathscr{S} \varphi_2) \mid \mathbf{A}(\varphi_1 \mathscr{B} \varphi_2). \tag{6}$$

The definition of $\mathscr{D}, s \models \varphi$ is augmented as follows.

- $\mathscr{D}, s \models \mathbf{A} \ominus(\psi)$ if for every t such that $(t, s') \models \rho$ we have $\mathscr{D}, t \models \psi$.
- $\mathscr{D}, s \models \mathbf{A}(\psi_1 \mathscr{S} \psi_2)$ iff for every run $s_0, s_1, \ldots$ and for every i such that $s = s_i$ there exists some $j \leq i$ such that $\mathscr{D}, s_j \models \psi_2$ and for all $j < k \leq i$ we have $\mathscr{D}, s_k \models \psi_1$.
- $\mathscr{D}, s \models \mathbf{A}(\psi_1 \mathscr{B} \psi_2)$ iff for every run $s_0, s_1, \ldots$ and for every i such that $s = s_i$ then for every $j \leq i$ we have $\mathscr{D}, s_j \models \psi_1$ or there is some $j < k \leq i$ such that $\mathscr{D}, s_k \models \psi_2$.

As before, and similar to LTL, we introduce the following abbreviations.

- $\mathbf{E} \ominus(\varphi) \equiv \neg \mathbf{A} \ominus(\neg\varphi)$,
- $\mathbf{E}(\varphi_1 \mathscr{S} \varphi_2) \equiv \neg \mathbf{A}(\neg\varphi_2 \mathscr{B} (\neg\varphi_1 \wedge \neg\varphi_2))$,
- $\mathbf{E}(\varphi_1 \mathscr{B} \varphi_2) \equiv \neg \mathbf{A}(\neg\varphi_2 \mathscr{S} (\neg\varphi_1 \wedge \neg\varphi_2))$.

Notice, that unlike in LTL, where the formula $\neg(\ominus \mathsf{true})$ identifies the initial location, the formula $\mathbf{E} \ominus(\mathsf{true})$ holds in initial states that have predecessors in $\mathscr{D}$. This is different from the treatment in [31], where it is assumed that initial states have no incoming transitions.

Our choice to have a branching past induces some strange consequences. For example, the formula $\mathbf{A} \square(\mathsf{grant} \rightarrow \mathbf{E} \diamondsuit(\mathsf{request}))$ says that every reachable state where grant is supplied is also reachable from a state where request is supplied and not necessarily on its actual past. On the other hand, $\mathbf{A} \square(\mathsf{grant} \rightarrow \mathbf{A} \diamondsuit(\mathsf{request}))$ says that all ways to reach grant must have a request on them. We note that choosing a linear past increases the complexity of CTL model checking to that of LTL. Additional studies of CTL with past are available in [35, 36].

We turn now to FOCTL, the extension of CTL to a logic that can reason about FDS with discrete variables that are not necessarily Boolean. As before, we generalize the set of propositions to a set of n-ary predicates for $n \geq 0$, denoted R. We use $f, g, \ldots$ to denote n-ary functions for $n \geq 0$ and $c, d, \ldots$ to denote constants. Let $var = \{x, y, \ldots\}$ be a countable set of variables. We define the set of terms (τ), atomic formulas (α), and formulas (φ).

$$\begin{array}{rcl} \tau & ::= & c \mid x \mid f(\tau, \ldots, \tau) \\ \alpha & ::= & r(\tau_1, \ldots, \tau_n) \mid \neg\alpha \mid \alpha_1 \vee \alpha_2 \mid \exists x.\alpha \\ \varphi & ::= & \alpha \mid \varphi_1 \vee \varphi_2 \mid \neg\varphi \mid \mathbf{A} \bigcirc(\varphi) \mid \mathbf{A}(\varphi_1 \mathscr{U} \varphi_2) \mid \mathbf{A}(\varphi_1 \mathscr{W} \varphi_2). \end{array} \tag{7}$$

Notice that the term $\bigcirc x$ that was included in the definition of FOLTL is removed here. As there may be multiple next states, it is not clear how to define the value of $\bigcirc x$.

A model for such a formula is an FDS $\mathscr{D} = \langle \mathscr{V}, \theta, \rho \rangle$, where every state $s \in \Sigma_{\mathscr{V}}$ has an interpretation for functions, constants, and variable symbols. Given such a model the semantics is defined as follows.

- For a term τ and a state s we define $s(\tau)$ as expected; for a variable x, $s(x)$ is given by the valuation in state s, for a constant c, $s(c)$ is given by the valuation in state s, and $s(f(\tau_1, \ldots, \tau_n))$ is $s(f)(s(\tau_1), \ldots, s(\tau_n))$.
- For an atomic formula α and a state s, we define when s satisfies α, denoted $s \models \alpha$, as follows.
  - $s \models r(\tau_1, \ldots, \tau_n)$ iff $s(r)(s(\tau_1), \ldots, s(\tau_n)) = \mathsf{true}$.
  - $s \models \neg\alpha$ iff $s \not\models \alpha$.
  - $s \models \alpha_1 \vee \alpha_2$ iff $s \models \alpha_1$ or $s \models \alpha_2$.
  - $s \models \exists x.\alpha$ iff there exists a model $\tilde{s}$ such that $\tilde{s}$ agrees with s on the interpretation of all functions, predicates, constants, and variables different from x such that $\tilde{s} \models \alpha$.

As in LTL, the main intention is to include references to variables ranging over discrete and infinite domains within a temporal context.

We mention informally CTL with past regular expressions. In this language, regular expressions are used to identify states that are reachable from an initial state with a computation that satisfies a given regular expression. This is the basis of the industrial specification language SUGAR that was succeeded by PSL (see [15] in this Handbook). We avoid defining this extension of CTL as it requires us to change the treatment of satisfaction from FDS to their computation trees, which we have not defined.

Finally, we discuss an extension of CTL that has to do with the ability to express fairness. The justice requirement can be stated in CTL as $\mathbf{A}\Box(\mathbf{A}\Diamond(p))$ (if every state has some successor). However, compassion cannot be expressed in CTL and CTL is not strong enough to demand that liveness properties hold only on fair paths. In order to solve this deficiency we add the path quantifiers $\mathbf{A}_f$ and $\mathbf{E}_f$, which range over fair paths. Formally, we extend the grammar in Eq. (5) by including $\mathbf{A}_f(\varphi_1 \mathscr{U} \varphi_2)$ and $\mathbf{A}_f(\varphi_1 \mathscr{W} \varphi_2)$ and consider FDS with fairness as models. The satisfaction of these formulas is defined over fair paths only:

- $\mathscr{D}, s \models \mathbf{A}_f(\varphi_1 \mathscr{U} \varphi_2)$ iff for every *fair path* $s_0, s_1, \ldots$ starting in s there is some i such that $s_i \models \varphi_2$ and for all $0 \leq j < i$ we have $s_j \models \varphi_1$.
- $\mathscr{D}, s \models \mathbf{A}_f(\varphi_1 \mathscr{W} \varphi_2)$ iff for every *fair path* $s_0, s_1, \ldots$ starting in s and for every $j \geq 0$ either $\mathscr{D}, s_j \models \varphi_1$ or there is $i \leq j$ such that $\mathscr{D}, s_i \models \varphi_2$.

As usual, we introduce the abbreviations $\mathbf{E}_f(\varphi_1 \mathscr{U} \varphi_2)$ for $\neg \mathbf{A}_f(\neg\varphi_2 \mathscr{W} (\neg\varphi_1 \wedge \neg\varphi_2))$ and $\mathbf{E}_f(\varphi_1 \mathscr{W} \varphi_2)$ for $\neg \mathbf{A}_f(\neg\varphi_2 \mathscr{U} (\neg\varphi_1 \wedge \neg\varphi_2))$. We note that there are some complications when the FDS has some runs that are not computations, i.e., if there are reachable states that have no successors. Further details are available in [13, 20]. This issue is revisited in the next section, where we consider various examples of the usage of LTL and CTL.

2.4.3 Model Checking and Satisfiability

When handling LTL, model checking and satisfiability are similar and indeed use almost the same techniques. Here, handling CTL, satisfiability is more complex than model checking and will use model checking as a subroutine. Thus, we start with studying model checking and then turn to satisfiability. Here we consider fairness-free FDSs as we do not discuss the extension to fair CTL.

Definition 12 (Model checking) Given an FDS $\mathscr{D}$ and a CTL formula φ, the model-checking problem for $\mathscr{D}$ and φ is to decide whether $\mathscr{D}$ implements φ.

Theorem 10 *CTL model checking is decidable in polynomial time.*

As before, the stated polynomial bound is in terms of the number of states of the FDS, which could be exponential in its representation. In order to solve the model-checking problem for a model $\mathscr{D}$ and a CTL formula φ we recursively compute the set of states that satisfy subformulas of φ.

Proof Consider an FDS $\mathscr{D} = \langle \mathscr{V}, \theta, \rho \rangle$ and a CTL formula φ. For every subformula ψ of φ we compute the set of states $[\![\mathscr{D}, \psi]\!]$. As before, we assume an efficient way to represent, manipulate, and compare assertions.

For a proposition p we have $[\![\mathscr{D}, p]\!] = p$.

For $\psi = \psi_1 \vee \psi_2$ we have $[\![\mathscr{D}, \psi]\!] = [\![\mathscr{D}, \psi_1]\!] \vee [\![\mathscr{D}, \psi_2]\!]$.

For $\psi = \neg\psi_1$ we have $[\![\mathscr{D}, \psi]\!] = \neg[\![\mathscr{D}, \psi_1]\!]$.

For $\psi = \mathbf{A} \bigcirc(\psi_1)$ we have $[\![\mathscr{D}, \psi]\!] = \neg\mathsf{pre}(\neg[\![\mathscr{D}, \psi_1]\!], \rho)$. That is, it is not the case that there is some successor that is not in $[\![\mathscr{D}, \psi_1]\!]$.

For $\psi = \mathbf{A}(\psi_1 \mathscr{U} \psi_2)$ Algorithm 5 computes $[\![\mathscr{D}, \psi]\!]$ (using fixpoints). When the fixpoint terminates, its value characterizes $[\![\mathscr{D}, \psi]\!]$. Intuitively, a state that is in $[\![\mathscr{D}, \psi_2]\!]$ definitely satisfies $\mathbf{A}(\psi_1 \mathscr{U} \psi_2)$. Otherwise, we gradually add states all of whose successors we know already satisfy $\mathbf{A}(\psi_1 \mathscr{U} \psi_2)$ and are also in $[\![\mathscr{D}, \psi_1]\!]$. We are careful not to include states in $[\![\mathscr{D}, \psi_1]\!]$ that have no successors.

Algorithm 5 Model check $\mathbf{A}(\psi_1 \mathscr{U} \psi_2)$

fix (new := $[\![\mathscr{D}, \psi_2]\!]$)
  new := new $\vee$ ($[\![\mathscr{D}, \psi_1]\!] \wedge \neg$pre($\neg$new, ρ) $\wedge$ pre(true, ρ));
end fix

Algorithm 6 Model check $\mathbf{A}(\psi_1 \mathscr{W} \psi_2)$

fix (new := true)
  new := new $\wedge$ ($[\![\mathscr{D}, \psi_2]\!] \vee ([\![\mathscr{D}, \psi_1]\!] \wedge \neg$pre($\neg$new, ρ)));
end fix

For $\psi = \mathbf{A}(\psi_1 \mathscr{W} \psi_2)$ Algorithm 6 computes $[\![\mathscr{D}, \psi]\!]$. When the fixpoint terminates, its value characterizes $[\![\mathscr{D}, \psi]\!]$. Intuitively, a state that is included in the fixpoint is either in $[\![\mathscr{D}, \psi_2]\!]$, in which case it clearly satisfies ψ, or it is in $[\![\mathscr{D}, \psi_1]\!]$ and all its successors are in the fixpoint as well. It follows that a maximal path starting at such a state either remains in the fixpoint forever (i.e., all states on the path satisfy ψ_1), or at some point reaches ψ_2 passing through states that satisfy ψ_1.

Finally, once the formula characterizing $[\![\mathscr{D}, \varphi]\!]$ is computed, we check whether $\theta \rightarrow [\![\mathscr{D}, \varphi]\!]$ to ensure that all initial states satisfy φ.

The number of stages of computation is related to the size of φ. Each and every one of the fixpoints is computed in time that is linear in the number of transitions of the given FDS. As before, the algorithm may be exponential in the size of the representation of the FDS. □

We now proceed to check satisfiability of CTL formulas. Given a CTL formula we construct a canonical FDS that checks whether it is satisfiable or not. This is called the *tableau* for φ. Essentially, we add Boolean variables for every subformula of φ and ensure that the transition relation satisfies the requirements of the subformulas. Unfortunately, this does not capture well the existential path formulas (negation of universal path formulas) and the fulfillment of until formulas (i.e., getting to a point where the second operand holds). So after building an initial FDS (referred to as a pre-tableau) we refine it (using Algorithm 7) to ensure that all formulas indeed hold. Algorithm 7 uses Algorithms 5 and 6 for model checking.

Assume that φ is converted to a formula that is syntactically derived from the grammar in Eq. (5). Let $\mathsf{cl}(\varphi)$ denote the set of subformulas of φ. Let $\mathscr{V}$ be the set of Boolean variables $\{v_\psi \mid \psi \in \mathsf{cl}(\varphi)\}$. We define the following essential consistency rules for these variables.

$$\mathsf{cons} \equiv \bigwedge_{\neg\psi \in \mathsf{cl}(\varphi)} (v_{\neg\psi} \leftrightarrow \neg v_\psi) \wedge \bigwedge_{\psi_1 \vee \psi_2 \in \mathsf{cl}(\varphi)} \big(v_{\psi_1 \vee \psi_2} \leftrightarrow (v_{\psi_1} \vee v_{\psi_2})\big).$$

Algorithm 7 Refine Pre-tableau

1: **fix** (new := true)
2: $\rho_{\mathsf{f}} := \rho \wedge \mathsf{new} \wedge \mathsf{prime}(\mathsf{new})$;
3: $\mathsf{new} := \mathsf{new} \wedge \bigwedge_{\mathbf{A}\bigcirc(\psi)\in\mathsf{cl}(\varphi)} (\neg v_{\mathbf{A}\bigcirc(\psi)} \rightarrow \mathsf{pre}(\neg v'_{\psi}, \rho_{\mathsf{f}}))$;
4: **for all** $(\mathbf{A}(\psi_1 \,\mathscr{U}\, \psi_2) \in \mathsf{cl}(\varphi))$
5: **fix** (until := new)
6: until := until $\wedge \neg v_{\psi_2} \wedge (\neg v_{\psi_1} \vee \mathsf{pre}(\mathsf{until}, \rho_{\mathsf{f}}) \vee \neg\mathsf{pre}(\mathsf{true}, \rho_{\mathsf{f}}))$;
7: **end fix**
8: $\mathsf{new} := \mathsf{new} \wedge (\neg v_{\mathbf{A}(\psi_1 \,\mathscr{U}\, \psi_2)} \rightarrow \mathsf{until})$;
9: **fix** (until := new $\wedge\, v_{\psi_2}$)
10: until := until $\vee\, (v_{\psi_1} \wedge \mathsf{new} \wedge \neg\mathsf{pre}(\neg\mathsf{until}, \rho_{\mathsf{f}}) \wedge \mathsf{pre}(\mathsf{true}, \rho_{\mathsf{f}}))$;
11: **end fix**
12: $\mathsf{new} := \mathsf{new} \wedge (v_{\mathbf{A}(\psi_1 \,\mathscr{U}\, \psi_2)} \rightarrow \mathsf{until})$;
13: **end for**
14: **for all** $(\mathbf{A}(\psi_1 \,\mathscr{W}\, \psi_2) \in \mathsf{cl}(\varphi))$
15: wuntil := backreach(new $\wedge \neg v_{\psi_1}, \neg v_{\psi_2} \wedge \rho_{\mathsf{f}}$);
16: $\mathsf{new} := \mathsf{new} \wedge (\neg v_{\mathbf{A}(\psi_1 \,\mathscr{W}\, \psi_2)} \rightarrow \mathsf{wuntil})$;
17: **end for**
18: **end fix**

Then a *pre-tableau* for φ is the FDS $\mathscr{D}_\varphi = \langle \mathscr{V}, \theta, \rho \rangle$, where θ and ρ are defined as follows.

$$
\begin{aligned}
&\theta = v_\varphi \wedge \mathsf{cons},\\
&\rho = \mathsf{cons} \wedge \mathsf{prime}(\mathsf{cons}) \qquad\qquad \wedge\\
&\qquad \bigwedge_{\mathbf{A}\bigcirc(\psi)\in\mathsf{cl}(\varphi)} v_{\mathbf{A}\bigcirc(\psi)} \rightarrow v'_{\psi} \qquad\qquad \wedge\\
&\qquad \bigwedge_{\mathbf{A}(\varphi_1 \,\mathscr{U}\, \varphi_2)\in\mathsf{cl}(\varphi)} v_{\mathbf{A}(\varphi_1 \,\mathscr{U}\, \varphi_2)} \rightarrow (v_{\varphi_2} \vee (v_{\varphi_1} \wedge v'_{\mathbf{A}(\varphi_1 \,\mathscr{U}\, \varphi_2)})) \wedge\\
&\qquad \bigwedge_{\mathbf{A}(\varphi_1 \,\mathscr{W}\, \varphi_2)\in\mathsf{cl}(\varphi)} v_{\mathbf{A}(\varphi_1 \,\mathscr{W}\, \varphi_2)} \rightarrow (v_{\varphi_2} \vee (v_{\varphi_1} \wedge v'_{\mathbf{A}(\varphi_1 \,\mathscr{W}\, \varphi_2)}))\,.
\end{aligned}
$$

We are now going to restrict the pre-tableau so that it respects the truth of existential formulas and delivery of until (so far until and weak-until are treated the same). Essentially, we compute the set of states that do not satisfy such formulas and effectively add conjuncts to θ and ρ (much like cons) that remove such states. Formally, we apply Algorithm 7 on the pre-tableau $\mathscr{D}_\varphi$.

The algorithm handles all existential path formulas and eventualities as follows. In line 3 we add the requirement that every subformula of the form $\mathbf{A}\bigcirc(\psi)$ behaves consistently. That is, if $v_{\mathbf{A}\bigcirc(\psi)}$ is false in a state, then some successor must have v_ψ false. In lines 5–8 we similarly handle consistency for subformulas of the form $\mathbf{A}(\psi_1 \,\mathscr{U}\, \psi_2)$. We compute the set of states that satisfy $\mathbf{E}(\neg\psi_2 \,\mathscr{W}\, (\neg\psi_1 \wedge \neg\psi_2))$ by

computing the fixpoint of $\mathsf{new} \wedge ((\neg\psi_1 \wedge \neg\psi_2) \vee (\neg\psi_2 \wedge \mathsf{pre}(\mathsf{new})) \vee (\neg\psi_2 \wedge \neg\mathsf{pre}(\mathsf{true})))$, which is simplified as above. Then, we require that if $v_{\mathbf{A}(\psi_1 \mathscr{U} \psi_2)}$ does not hold, then the state satisfies the computed assertion new. A similar consistency requirement is added for subformulas of the form $\mathbf{A}(\psi_1 \mathscr{W} \psi_2)$ in lines 15–16. Finally, lines 9–12 compute the set of states that satisfy $\mathbf{A}(\psi_1 \mathscr{U} \psi_2)$ and add a positive guarantee that when $v_{\mathbf{A}(\psi_1 \mathscr{U} \psi_2)}$ holds then it is the case that $\mathbf{A}(\psi_1 \mathscr{U} \psi_2)$ indeed holds. The external fixpoint ensures that the effect of removed states and transitions is further propagated backward until stabilization. This ensures that all eventualities and existential path properties are indeed fulfilled. Finally, a tableau for φ is $\mathscr{T}_\varphi = \langle \mathscr{V}, \theta \wedge \mathsf{new}, \rho_\mathsf{f} \rangle$. We are now ready to define and decide satisfiability.

Definition 13 (Satisfiability) A CTL formula φ is satisfiable if some FDS implements it. That is, φ is satisfiable if there is an FDS $\mathscr{D}$ such that $\mathscr{D} \models \varphi$.

Theorem 11 *CTL satisfiability is decidable in exponential time.*

Proof If in $\mathscr{T}_\varphi$ the initial condition is not equivalent to false we conclude that φ is satisfiable.

It is simple to see by construction of $\mathscr{T}_\varphi$ that if v_ψ holds in a state s of $\mathscr{T}_\varphi$ then $\mathscr{T}_\varphi, s \models \psi$. In particular, the initial condition requires that v_φ holds, thus if the initial condition is not false we have found an FDS that satisfies φ.

Showing that if φ is satisfiable then the above construction will supply an FDS that satisfies φ is more complicated. The proof proceeds by taking an arbitrary FDS $\mathscr{D}$ implementing φ (including infinite state) and constructing a quotient $\mathscr{D}/_\equiv$. Two states, s and t, are considered equivalent $s \equiv t$ iff they agree on the truth value of all formulas in $\mathsf{cl}(\varphi)$. Then, $\mathscr{D}/_\equiv$ is obtained from $\mathscr{D}$ by considering equivalence classes as new states. Two equivalence classes $[s]$ and $[t]$ are connected if $\rho(s,t)$ in $\mathscr{D}$. Unfortunately, the quotient operation may introduce loops that interfere with satisfaction of until formulas (either of the type $\mathbf{A}(\psi_1 \mathscr{U} \psi_2)$ or of the type $\neg\mathbf{A}(\psi_1 \mathscr{W} \psi_2)$). So the quotient structure needs to be pruned to ensure that all eventualities are fulfilled. For a full account we refer the reader to [18].

We note that the number of states of the pre-tableau is exponential in the size of φ. Thus, checking that until subformulas hold on the pre-tableau may require exponential time and lead to the stated time bound. □

2.5 Examples for LTL and CTL

In this section we explore the usage of LTL and CTL. We give examples of specifications and discuss their meaning. We also start hinting at the differences between LTL and CTL and their respective advantages and disadvantages. As our definition of LTL considers only infinite paths we assume that FDSs considered in this section have no finite paths.

2.5.1 Invariance and Safety

We start with the simplest type of property. An invariance simply says that there is an assertion that holds over all reachable states of the program. Consider Peterson's mutual exclusion algorithm represented in Fig. 2. We would like to establish that no matter what, the two processes will not visit their respective critical sections at the same time. The assertion that characterizes states where both systems are in their critical section is $\mathsf{at}_{l_5} \wedge \mathsf{at}_{m_5}$. Then, $\Box \neg(\mathsf{at}_{l_5} \wedge \mathsf{at}_{m_5})$ in LTL and $\mathbf{A}\Box(\neg(\mathsf{at}_{l_5} \wedge \mathsf{at}_{m_5}))$ in CTL say exactly that: it globally holds, i.e., in every reachable state, that the two programs are not both in their critical sections. The same property for the semaphore example in Fig. 3 is written $\Box \neg(\mathsf{at}_{l_3} \wedge \mathsf{at}_{m_3})$ in LTL and $\mathbf{A}\Box(\neg(\mathsf{at}_{l_3} \wedge \mathsf{at}_{m_3}))$ in CTL.

Invariants can also be used to specify that some locations in a program are never reached. Consider for example the case of a software program that includes an assertion. Then, we can replace this assertion by a conditional statement and check an invariant saying that the location inside the condition is never reached. Invariants are relatively easy to check as they just require the computation of the set of reachable states. Furthermore, if exploring the states walking forward (using reach and not backreach) then the exploration can be stopped as soon as a violation is found.

An interesting property is a *transition invariant*. While invariants must hold for all reachable states, transition invariants must hold for all reachable transitions. In general, they can be written as $\Box(\alpha)$ or $\mathbf{A}\Box(\alpha)$, where α allows Boolean operators and at most one level of nesting of the $\bigcirc$ (or $\mathbf{A}\bigcirc$) operator. For example, the *unless* property of unity [11] is such a transition invariant. Whenever assertion p holds and q does not hold, then the next state must satisfy p or q. Formally, $\Box((p \wedge \neg q) \rightarrow \bigcirc(p \vee q))$ in LTL and $\mathbf{A}\Box((p \wedge \neg q) \rightarrow \mathbf{A}\bigcirc(p \vee q))$ in CTL. We note that the same property can be written as $\Box(p \rightarrow p \mathscr{W} q)$, which may be clearer, and similarly in CTL. Another transition invariant is the property of *stabilization*. Whenever assertion p becomes true it remains true forever. Formally, $\Box(p \rightarrow \Box p)$ or $\mathbf{A}\Box(p \rightarrow \mathbf{A}\Box(p))$ can be written also as $\Box(p \rightarrow \bigcirc p)$ or $\mathbf{A}\Box(p \rightarrow \mathbf{A}\bigcirc(p))$. Transition invariants are important as they are relatively easy to check. Computation of reachable states inevitably has to take transitions into account. Thus, checking of transition invariants can be easily integrated into reachability analysis.

The formula $\Box(\mathsf{grant} \rightarrow \diamondsuit \mathsf{request})$ states the precedence in time of request over grant. It is equivalent to $(\neg\mathsf{grant}) \mathscr{W} \mathsf{request}$ and can be expressed in CTL as $\mathbf{A}(\neg\mathsf{grant} \mathscr{W} \mathsf{request})$. The stronger property $\Box(\mathsf{grant} \rightarrow \ominus(\neg\mathsf{grant} \mathscr{S} \mathsf{request}))$ states that every grant comes after a previous request and that once a grant has been given no additional grant can be given without a new request. The property

$$\Box\big(\mathsf{grant}_2 \wedge \diamondsuit \mathsf{grant}_1 \rightarrow \big(\neg\mathsf{request}_1 \mathscr{S} (\mathsf{request}_2 \wedge \diamondsuit \mathsf{request}_1)\big)\big)$$

states that whenever grant_2 comes after grant_1, the respective requests happened in the same order.

Similarly, the property $\Box(\mathsf{light} \rightarrow \mathsf{light} \mathscr{S} \mathsf{call})$ [4], taken from a specification of a lift system, says that if the light is on it has been on continuously since a call

was made. The same property can be stated without usage of past operators as $\Box(\bigcirc \mathsf{light} \rightarrow (\mathsf{light} \vee \mathsf{call}))$. Here we assume that light and call cannot occur together.

Finally, we mention that the two properties $p \mathbin{\mathscr{W}} (q \mathbin{\mathscr{W}} r)$ and $(p \mathbin{\mathscr{W}} q) \mathbin{\mathscr{W}} r$ are not equivalent. Indeed, the first requires that p holds continuously either forever or up to a point where $q \mathbin{\mathscr{W}} r$ starts holding. The second requires that $p \mathbin{\mathscr{W}} q$ hold continuously either forever or until r. In particular, the satisfaction of $p \mathbin{\mathscr{W}} q$ may need to be checked after the occurrence of r, which may be slightly less clear. For example, a sequence of p's followed by an r (and neither p nor q) satisfies the first but not the second.

2.5.2 Liveness

We now proceed to present some liveness properties. The simplest liveness property is $\Diamond p$ for some atomic proposition p. For example, consider the program in Fig. 1. Termination of this program is expressed as $\Diamond \mathsf{at}_{l_3}$. We note that this property does not hold for the program as presented as an FDS in Sect. 2.2. Indeed, the program includes the stuttering clause allowing the program to stay in every one of its states forever. If we add the justice requirements $\mathscr{J} = \{\neg \mathsf{at}_{l_i} \mid i \in \{0, 1, 2\}\}$ then every computation does satisfy the property. The same property in CTL is $\mathbf{A} \Diamond (\mathsf{at}_{l_3})$. As in LTL, it does not hold over this system. As this program runs in isolation, we can "factor in" the effects of justice. By removing the stuttering clause from the transitions of the system we get a system that does satisfy this property. Another option is to use fairness and replace $\mathbf{A} \Diamond (\mathsf{at}_{l_3})$ by $\mathbf{A}_f \Diamond (\mathsf{at}_{l_3})$.

We turn our attention to Peterson's mutual exclusion algorithm (Fig. 2). Here, we would like to establish that each process can access the critical section. Without loss of generality, we consider only the left-hand process. The case of the other process is identical. In LTL this is expressed as $\Box(\mathsf{at}_{l_2} \rightarrow \Diamond \mathsf{at}_{l_5})$. Namely, whenever the left-hand-side process leaves the non-critical section and starts the process of reaching the critical section it will eventually get there. As before, to express the same in CTL we have to either remove all stuttering—leaving the option to stay in the noncritical section—or use fair path quantification. Here, we could use either $\mathbf{A}_f \Box(\mathsf{at}_{l_2} \rightarrow \mathbf{A}_f \Diamond(\mathsf{at}_{l_5}))$ or $\mathbf{A} \Box(\mathsf{at}_{l_2} \rightarrow \mathbf{A}_f \Diamond(\mathsf{at}_{l_5}))$. Accessibility in the program using semaphores in Fig. 3 is $\Box(\mathsf{at}_{l_2} \rightarrow \Diamond \mathsf{at}_{l_3})$. The same property in CTL is $\mathbf{A} \Box(\mathsf{at}_{l_2} \rightarrow \mathbf{A} \Diamond(\mathsf{at}_{l_3}))$. This time, as the program uses compassion, the only way to reason about this property is by using fair path quantification as in $\mathbf{A} \Box(\mathsf{at}_{l_2} \rightarrow \mathbf{A}_f \Diamond(\mathsf{at}_{l_3}))$.

In general the property of justice—that an assertion is true infinitely often along a computation—is expressible in both LTL and CTL. Given a justice requirement J, writing $\Box \Diamond J$ in LTL or $\mathbf{A} \Box(\mathbf{A} \Diamond(J))$ in CTL has exactly the same meaning. However, this property checks whether all paths of the program are fair. When checking liveness properties, we often rely on fairness to make progress. Thus, the property is only going to hold on fair paths. This is easily expressed in LTL by writing $\Box \Diamond J \rightarrow \varphi$, where φ is the liveness property we are interested in. However,

$\mathbf{A}\,\square(\mathbf{A}\,\diamondsuit(J)) \rightarrow \psi$, even if ψ is the CTL equivalent of φ, says that only if *all* paths of the program are fair do we expect ψ to hold. This is the reason to introduce the fair path quantifier.

A similar issue arises when we consider assumptions about the behavior of the program. In many cases, model checking is applied to part of a program, or a program that is expected to be used in a given environment. This is very similar to fairness, where we assume that some scheduler is going to handle the many processes in some fair way but do not model this scheduler explicitly. In LTL such a situation is easily handled by using implication. Indeed, $\varphi \rightarrow \psi$ says exactly that only computations that satisfy φ should satisfy ψ. If φ is some environment assumption, then we get that ψ should hold only on paths where the environment is well behaved. Consider for example Peterson's mutual exclusion algorithm in Fig. 2. Using assume-guarantee we could analyze just one of the components of the program. Indeed, if we add the assumption that $\square(t = 1 \rightarrow \diamondsuit\, t = 0)$ or that $\square\diamondsuit(y = 0)$, then every environment that changes y and t arbitrarily but ensures either of these properties will guarantee accessibility. Thus, the left-hand-side process fulfills $\square(t = 1 \rightarrow \diamondsuit\, t = 0) \rightarrow \square(\mathsf{at}_{l_2} \rightarrow \diamondsuit\, \mathsf{at}_{l_5})$. Notice, however, that we have to add an environment that changes t and y arbitrarily when the system is not changing them. Similarly, for the semaphores example in Fig. 3 it is enough to assume that the environment releases x infinitely often. Thus, the left-hand-side system satisfies $(\square\diamondsuit x = 1) \rightarrow \square(\mathsf{at}_{l_2} \rightarrow \diamondsuit\, \mathsf{at}_{l_3})$. Notice, that the environment can increase the value of x even when the process owns x. This will obviously violate the safety of the protocol but will guarantee its liveness. Notice, that the assumptions have to be discharged in an acyclic way. For example, in order to complete the proof of accessibility for Peterson we have to show that the environment guarantees $\square(t = 1 \rightarrow \diamondsuit\, t = 0)$ and do that without relying on accessibility of the left-hand-side process. Circular reasoning in the context of liveness usually does not work. One simple strategy to avoid cyclic reasoning is by having an order on the components of the system that supply assumptions. CTL does not support a simple implementation of assume-guarantee.

2.5.3 Additional Examples

So far most examples were equally easily expressed in LTL and CTL. We now explore a few specifications that show the differences between the two.

We start with an example that presents the exact meaning of the tight pairing of path quantifiers and temporal operators in CTL. Consider the two CTL properties, $\mathbf{A}\diamondsuit(\mathbf{A}\bigcirc(p))$ and $\mathbf{A}\bigcirc(\mathbf{A}\diamondsuit(p))$. The two are different. Indeed, in the first, every path of the system must reach a state all of whose successors satisfy p. On the other hand in the second, it is enough that every path of the system reaches a state that satisfies p. Thus, the first requires the p's to be all successors of the same state and the second does not. Interestingly, the second property is expressible in LTL as either $\bigcirc\diamondsuit p$ or $\diamondsuit\bigcirc p$. The property $\mathbf{A}\diamondsuit(\mathbf{A}\bigcirc(p))$ cannot be expressed in LTL.

Consider the property $\mathbf{A}\,\Box(\mathsf{withdrawal} \rightarrow (\mathbf{A}\,\bigcirc(\mathsf{success}) \vee \mathbf{A}\,\bigcirc(\neg\mathsf{success})))$ [10]. It states that following a withdrawal request either in all possible futures the request is successful or in all possible futures the request is unsuccessful. Clearly, this property is not expressible in LTL.

The problem of *deadlock*, when all processes in a system are stuck, or rather its avoidance, is a very important property of concurrent systems. This can be expressed in CTL as $\mathbf{A}\,\Box(\mathbf{E}\,\bigcirc(\mathsf{true}))$. That is, every state has some successor. Clearly, we have to disallow stuttering in order for the check of this property to be meaningful. In LTL we restrict attention to infinite computations and avoid the issue of deadlocks. It is simple to incorporate search for deadlocks when computing the set of reachable states. However, the LTL model-checking algorithm calls for the composition of an FDS with a temporal tester. Even if the FDS does not have deadlocks, its composition with the temporal tester may have deadlocks (due to nondeterminism and deadlocks of the temporal tester). Thus, in order to check for deadlocks in the LTL framework we have to implement it separately.

An interesting property, of a similar flavor, showing the power of CTL is that of *recovery*. Essentially, it says that a system can always recover. Suppose that start is an assertion characterizing some states in which the system can start again. A property like $\mathbf{A}\,\Box(\mathbf{E}\,\Diamond(\mathsf{start}))$ states that from every reachable state of the program, it is possible to get to a state satisfying start. Having a *reset sequence* that gets a program to a safe initial state is important in hardware design. Recovery is also an assumption of black-box checking [40]. Clearly, this combination of universal and existential path quantification is impossible to express in LTL.

2.6 CTL*

As demonstrated in Sect. 2.5, LTL and CTL are different. However, in order to compare them formally, we have to reason about the same sets of models. In order to do that, we think about both LTL and CTL as characterizing sets of FDSs. We then show that LTL and CTL are incomparable. We exhibit families of models that can be distinguished by LTL formulas but cannot be distinguished by CTL formulas and vice versa. This leads to the definition of CTL*, an expressive logic that combines both LTL and CTL. Effectively, it combines the full LTL with the path quantifiers introduced in CTL.

2.6.1 Branching vs. Linear Time

As explained, models of LTL formulas are infinite sequences and models of CTL formulas are FDSs. In order to be able to compare the two, we consider the definition of implementation for both logics. We show that some formulas in LTL cannot be expressed in CTL and some formulas in CTL cannot be expressed in LTL. That is,

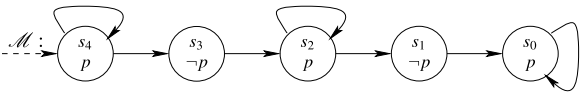


Fig. 5 Systems $\mathscr{M}_i$ that satisfy $\diamondsuit\square p$

for an LTL/CTL formula φ let $\mathsf{Imp}(\varphi)$ denote the set of implementations of φ. Then, there is an LTL formula φ such that for every CTL formula ψ we have $\mathsf{Imp}(\varphi) \neq \mathsf{Imp}(\psi)$, and vice versa.

We start by showing that LTL can express properties that cannot be expressed in CTL. Specifically, the formula $\diamondsuit\square p$ cannot be expressed in CTL. As we explain below, the natural CTL candidate to express the same property is $\mathbf{A}\diamondsuit(\mathbf{A}\square(p))$. However, the latter requires that p starts holding in all futures simultaneously, which is more than the LTL formula stipulates. We define a family of systems $\mathscr{M}$ such that $\diamondsuit\square p$ holds over $\mathscr{M}$. We show that every CTL formula that holds over all the systems in $\mathscr{M}$ must hold also over a system that falsifies $\diamondsuit\square p$. The system is depicted in Fig. 5. For convenience, we depict all the family $\mathscr{M}$ as one infinite-state system, however, every instance in the family includes a finite number of states. Formally, using $a \dot{-} b \equiv \mathsf{max}(0, a - b)$ we set $\mathscr{M}_i = \langle\{p, y\}, p \wedge y = i, \rho_i\rangle$, where p is a Boolean variable and y ranges over $\{0, \ldots, i\}$ and ρ_i is defined as follows.

$$\rho_i \equiv \big(p \rightarrow y = y'\big) \wedge \big(\neg p \rightarrow y' = y \dot{-} 1\big) \wedge \big((p \wedge y > 0) \vee p'\big).$$

It is simple to see that for $j \geq 1$ state s_{2j-1} corresponds to the valuation $y = j$ and $\neg p$ and for $j \geq 0$ state s_{2j} corresponds to the valuation $y = j$ and p. Also, every infinite path eventually remains in state s_{2j} for some j. Hence, every infinite path satisfies $\diamondsuit\square p$ and every system $\mathscr{M}_i$ satisfies the LTL formula $\diamondsuit\square p$.

We now show that this cannot be the case for a CTL formula.

Lemma 7 *For every CTL formula φ such that for all $i \geq 0$ we have $\mathscr{M}_i, t_{2i} \models \varphi$ there is a system $\mathscr{N}$ such that $\mathscr{N} \models \varphi$ and $\mathscr{N} \not\models \diamondsuit\square p$.*

Proof Let n be the number of subformulas of φ and let m denote 2^n. Consider the system $\mathscr{M}_m$. We are going to identify two states t_{2i} and t_{2j} for $i < j \leq m$ such that the sets of subformulas of φ that hold in t_{2i} and t_{2j} are identical. Furthermore, we are going to identify a path between t_{2i} and t_{2j} such that all universal eventualities that should be true in t_{2i} are fulfilled before arriving at t_{2j} and all existential eventualities that should be true in t_{2i} are fulfilled on paths that diverge from this identified path. Then, we create a modified system where this specific path between t_{2i} and t_{2j} is closed to form a loop. Clearly, this path falsifies the LTL formula $\diamondsuit\square p$. However, from the construction of this path, all subformulas of φ that hold in t_{2m} still hold in the modified system. In particular, φ holds in t_{2m} showing that φ cannot be equivalent to $\diamondsuit\square p$.

We modify the system $\mathscr{M}_m$ as follows. Consider the state t_{2m} and let C_m be the set of subformulas of φ that hold in t_{2m}. Consider a subformula $\psi \in C_m$ of the form $\psi = \mathbf{A}(\psi_1 \,\mathscr{U}\, \psi_2)$. As ψ holds in t_{2m} it must be the case that ψ_2 holds in t_{2m}, otherwise ψ does not hold on the path $t_{2m}, t_{2m}, t_{2m}, \ldots$. Consider the set of

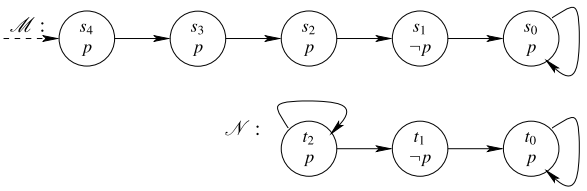


Fig. 6 Systems $\mathscr{M}_i$ that satisfy $\mathbf{A}\diamondsuit(\mathbf{A}\square(p))$ and system $\mathscr{N}$ that does not

subformulas of the form $\mathbf{E}\bigcirc(\psi_1)$ or $\mathbf{E}(\psi_1\,\mathscr{U}\,\psi_2)$ that hold in t_{2m}. There is a finite set of paths that start in t_{2m} and show satisfaction of all existential path formulas. Of all these paths, there is a maximal number k_m of repetitions of the state t_{2m} on a path. Let $\mathscr{N}_m$ denote the system that is obtained from $\mathscr{M}_m$ by replacing the state t_{2m} by a chain of states $t_{2m}^1,\ldots,t_{2m}^{k_m}$ such that t_{2m}^j is connected to t_{2m}^{j-1} and to a copy of $\mathscr{M}_{m-1}$. Clearly, all formulas of C_m still hold over $\mathscr{N}_m$ and all eventualities that are promised in t_{2m}^1 are fulfilled before arriving at $t_{2m}^{k_m}$.

We now modify $\mathscr{N}_m$ by changing the copy of t_{2m-2} that is connected to $t_{2m}^{k_m}$ to create $\mathscr{N}_{m-1}$ by the same process. We repeat this process until at some point, we find that the set of subformulas of φ that hold in t_{2i} is equivalent to C_m. Then, we simply connect t_{2m}^1 instead of t_{2i} and create a loop.

The modified system still satisfies all CTL formulas that are promised to hold in C_m and in particular φ. □

Our proof is based on Rabin's result about expressiveness of tree automata [46]. An alternative proof based on systems with fairness constraints is available in [12].

Corollary 2 *CTL is not as expressive as LTL.*

We now show that LTL is not as expressive as CTL, establishing the two as incomparable. It is simple to find a formula that combines universal and existential path quantification, such as $\mathbf{A}\square(\mathbf{E}\diamondsuit(p))$, that cannot be expressed in LTL. It is more interesting to find a formula that uses only universal path quantifiers and is not expressible in LTL. We prove the dual of Lemma 7 and show that $\mathbf{A}\diamondsuit(\mathbf{A}\square(p))$ cannot be expressed in LTL. We define a family of systems that satisfy $\mathbf{A}\diamondsuit(\mathbf{A}\square(p))$ and a system that does not satisfy $\mathbf{A}\diamondsuit(\mathbf{A}\square(p))$. Consider the family of systems $\mathscr{M}$ and the system $\mathscr{N}$ depicted in Fig. 6. For convenience, we depict all the family $\mathscr{M}$ as one infinite-state system. Formally, we set $\mathscr{M}_i = \langle\{p, y\}, \theta_i, \rho_i\rangle$, where p is a Boolean variable and y ranges over $\{0,\ldots,i\}$ and θ_i and ρ_i are defined as follows.

$$\begin{aligned}\theta_i &\equiv (y = i) \wedge (p \leftrightarrow i \neq 1),\\ \rho_i &\equiv \big(p' \leftrightarrow i \neq 2\big) \wedge \big(y' = y \dot{-} 1\big).\end{aligned}$$

For $j \geq 1$ state s_j corresponds to the valuation $y = j$. It is simple to see that for $1 \leq j \leq i$ we have $\mathscr{M}_i, s_j \models \mathbf{A}\diamondsuit(\mathbf{A}\square(p))$. We set $\mathscr{N} = \langle\{p, x\}, \rho, p \wedge x\rangle$, where

p and x are Boolean variables and ρ is defined as follows.

$$\rho \equiv \big(p \rightarrow \big(x' = x\big)\big) \wedge \big(\neg p \rightarrow \neg x'\big) \wedge \big((p \wedge x) \vee p'\big)$$

State t_2 corresponds to $x \wedge p$, state t_1 corresponds to $x \wedge \neg p$, and state t_0 corresponds to $\neg x \wedge p$. The figure does not depict the unreachable state $\neg x \wedge \neg p$. It is simple to see that $\mathscr{N}, t_2 \not\models \mathbf{A} \diamondsuit (\mathbf{A} \square (p))$.

Lemma 8 *For every LTL formula φ such that for all $i \geq 0$ we have $\mathscr{M}_i, t_i \models \varphi$ it holds that $\mathscr{N}, t_2 \models \varphi$.*

Proof By assumption φ holds over $\mathscr{M}_i, t_i$. For every $i > 0$, the label of the unique computation of $\mathscr{M}_i$ that starts in t_i is $\{p\}^i \emptyset \{p\}^\omega$ and the computation of $\mathscr{M}_0$ that starts in t_0 is $\{p\}^\omega$. Let W be the set of all such models. That is,

$$W = \big\{\{p\}^\omega, \{p\}^i \emptyset \{p\}^\omega \;\big|\; i > 0\big\}.$$

It follows that for every word $w \in W$ we have $w \models \varphi$.

However, the set of paths of $\mathscr{N}$ is exactly W. Hence, it must be the case that $\mathscr{N} \models \varphi$. □

Corollary 3 *LTL is not as expressive as CTL.*

The differences between model checking CTL and LTL justify further studies into which formulas are expressible in both. An initial result analyzed which CTL formulas are expressible in LTL. It established that a CTL formula is expressible in LTL if and only if it can be expressed in LTL by removing all path quantifiers [12]. Formally, given a CTL formula φ let φ^d denote the LTL formula that is obtained from φ by removing (syntactically) all the path quantifiers.

Theorem 12 *The formula φ is expressible in LTL iff $\varphi = \varphi^d$.*

Given this correspondence and the interpretation of LTL as holding over all paths in an FDS, it makes sense to concentrate on universal path quantification in CTL. This motivates the introduction of the sub-logic ACTL. Formally, ACTL is the logic obtained from CTL by disallowing the use of universal path quantifiers under an odd number of negations. Equivalently, we can add $\wedge$ to the syntax of CTL and allow negation only to be applied to propositions. There is a good characterization of ACTL formulas that can be expressed in LTL [38]. Such ACTL formulas are called deterministic. Formally, propositional formulas and ACTL^{det} formulas are defined by the following grammar:

$$\begin{aligned} &\alpha \;::=\; p \mid \neg\alpha \mid \alpha_1 \vee \alpha_2, \\ &\varphi \;::=\; \alpha \mid (\alpha \wedge \varphi_1) \vee (\neg\alpha \wedge \varphi_2) \mid \varphi_1 \wedge \varphi_2 \mid \mathbf{A} \bigcirc (\varphi) \mid \\ &\mathbf{A}\big((\alpha \wedge \varphi_1)\, \mathscr{U}\, (\neg\alpha \wedge \varphi_2)\big) \mid \mathbf{A}\big((\alpha \wedge \varphi_1)\, \mathscr{W}\, (\neg\alpha \wedge \varphi_2)\big). \end{aligned} \tag{8}$$

Intuitively, ACTL$^{\det}$ has no "real" choices. Operators that display choice are $\vee$, $\mathscr{U}$, and $\mathscr{W}$. In these operators *the same* propositional formula is conjuncted positively on one side and negatively on the other. So the valuation of propositions in a state dictates which branch of the choice applies.

Theorem 13 *An ACTL formula is expressible in LTL iff it is in ACTL^{det}.*

Furthermore, the language of the equivalent LTL formula (which according to Theorem 12 is obtained by removing the path quantifiers) can be expressed as the complement of a finite union of sets of the following form.

$$\Sigma_1^* \cdot w_1 \cdot \Sigma_2^* \cdot w_2 \ldots w_{n-1} \cdot \Sigma_{n-1}^* \cdot w_n \cdot \Sigma_n^\omega,$$

where w_i is a letter over the propositions appearing in the formula and Σ_i is a set of letters over the propositions appearing in the formula. For further details we refer the reader to [38].

However, it turns out that there are CTL formulas that are expressible in LTL that are not ACTL formulas [7]. Thus, it may be necessary to use existential path quantification in order to express universal properties expressible in both CTL and LTL.

Theorem 14 *There is an LTL formula expressible in CTL but not in ACTL.*

For an example of such an LTL formula and further details we refer the reader to [7]. We note that a full characterization of which CTL formulas are expressible in LTL is unknown.

2.6.2 CTL* Definition

The mutual deficiencies of LTL and CTL motivated the introduction of a logic that encompasses both. Essentially, CTL* combines LTL's unrestricted nesting of temporal operators with CTL's universal and existential path quantification. The resulting logic, called CTL*, is a very expressive temporal logic that includes both. We now define CTL*.

As before, we assume a countable set of Boolean propositions P. Models for CTL*, as for CTL, are fairness-free FDSs. Here we assume that the FDS has no dead ends. If $\hat{P}$ is the set of propositions appearing in φ, then every FDS $\mathscr{D}$ such that $\hat{P} \subseteq \mathscr{V}$ is a model for φ.

We define state formulas (φ) and path formulas (ψ) as follows. CTL* formulas are state formulas.

$$\begin{aligned} \varphi &::= p \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \mathbf{A}\psi, \\ \psi &::= \varphi \mid \neg\psi \mid \psi_1 \vee \psi_2 \mid \bigcirc\psi \mid \psi_1 \mathscr{U} \psi_2. \end{aligned} \tag{9}$$

The truth of state formulas is interpreted over states, thus, as in CTL, we define the set of states that satisfy such a formula. The truth of path formulas is interpreted over paths, thus, as in LTL, we define the set of paths that satisfy such a formula.

Formally, for a formula φ and an FDS $\mathscr{D} = \langle \mathscr{V}, \theta, \rho \rangle$, we say that φ *holds in state s of* $\mathscr{D}$, written $\mathscr{D}, s \models \varphi$ and define it inductively as follows (omitting Boolean connectives).

- For $p \in P$ we have $\mathscr{D}, s \models p$ iff $s \models p$.
- $\mathscr{D}, s \models \mathbf{A}\psi$ iff for every path $\sigma = s_0, s_1, \ldots$ starting in state s we have $\sigma, 0 \models \psi$.
- For path formulas we define satisfaction over paths as follows.
  - $\sigma, i \models \varphi$ iff $\mathscr{D}, s_i \models \varphi$, for a state formula φ.
  - $\sigma, i \models \bigcirc \psi$ iff $\sigma, i+1 \models \psi$.
  - $\sigma, i \models \psi_1 \mathscr{W} \psi_2$ iff there exists $k \geq i$ such that $\sigma, k \models \psi_2$ and $\sigma, j \models \psi_1$ for all $j, i \leq j < k$.

We can use the same abbreviations as previously for existential path quantification $\mathbf{E}$ and the temporal operators $\diamondsuit$, $\square$, and $\mathscr{W}$.

As in CTL, given a state formula φ and an FDS $\mathscr{D}$, we denote by $[\![\mathscr{D}, \varphi]\!] = \{s \in \Sigma_{\mathscr{V}} \mid \mathscr{D}, s \models \varphi\}$.

Definition 14 (Implementation) We say that an FDS $\mathscr{D}$ *implements* specification φ, denoted $\mathscr{D} \models \varphi$, if every initial state of $\mathscr{D}$ satisfies the formula.

We note that, syntactically, CTL* includes both CTL and (future) LTL. Here we have chosen not to include past operators in CTL*. In order to include past operators, we would include in the universal path quantification all the possible points in paths that pass through a state s.

*2.6.3 Examples of Usage of CTL**

One of the main deficiencies of CTL is its inability to express fairness and the lack of support for assume guarantee. Clearly, CTL* solves the two deficiencies. Regarding fairness, in CTL* the path quantifier $\mathbf{A}_f$ can be expressed within the logic. Given justice and compassion $\mathscr{J}$ and $\mathscr{C}$, the formula

$$\mathbf{A}\left(\left(\bigwedge_{J \in \mathscr{J}} \square \diamondsuit J \wedge \bigwedge_{\langle P, Q \rangle \in \mathscr{C}} (\square \diamondsuit P \rightarrow \square \diamondsuit Q)\right) \rightarrow \varphi\right)$$

is equivalent to $\mathbf{A}_f \varphi$. Furthermore, φ can nest temporal operators freely. This way, CTL* can combine quantification on all paths and on some paths in the same formula. Similarly, the assume-guarantee framework works in CTL* just as simply as it does in LTL. Also, assumptions can be restricted to specific parts of the specification.

CTL* is strong enough to act similarly with respect to properties expressed in CTL. Recall the property of recovery $\mathbf{A}\Box(\mathbf{E}\Diamond(\mathsf{start}))$. CTL* can express the property that every computation that maintains recoverability will actually recover $\mathbf{A}(\Box(\mathbf{E}\Diamond(\mathsf{start})) \rightarrow \Diamond\,\mathsf{start})$, which is not expressible either in LTL or in CTL.

Finally, CTL* is strong enough to express the problem of realizability of LTL specifications (see elsewhere in this Handbook [6] (Bloem et al., Graph Games and Reactive Synthesis)). A system is called *open* with respect to a set of variables $\mathscr{X}$ if for every state s and for every valuation $\hat{t}$ of $\mathscr{X}$ there is a successor t of s such that $\hat{t} = t \Downarrow_{\mathscr{X}}$. Realizability is essentially the question of existence of open implementations. Clearly, this question can be expressed in CTL*. For every possible valuation v of $\mathscr{X}$ the formula $\mathbf{E}\bigcirc(v)$ requires that a state has a successor with valuation v. Then $\mathsf{open} \equiv \mathbf{A}\Box(\bigwedge_{v\in\Sigma_{\mathscr{X}}} \mathbf{E}\bigcirc(v))$ requires that a system be open. Finally, given an LTL formula φ, the formula $\mathsf{open} \wedge \mathbf{A}\varphi$ is satisfiable exactly iff there is an open system satisfying φ.

2.6.4 Model Checking and Satisfiability

We now turn to model checking of CTL*, which, like LTL, can be solved in polynomial space [13]. The model-checking algorithm we present is a combination of the LTL algorithm and the CTL algorithm. Essentially, just like in CTL we compute the set of states satisfying a formula by first starting from simpler formulas. In order to compute the set of states satisfying the path formula we invoke the LTL model checking algorithm. Then, using the set of states satisfying a path formula we treat the computed assertion as a new proposition and continue with the recursive treatment as in CTL.

Definition 15 (Model checking) Given an FDS $\mathscr{D}$ and a CTL* formula φ, the model-checking problem for $\mathscr{D}$ is to decide whether $\mathscr{D}$ implements φ.

Theorem 15 *CTL* model checking is decidable in polynomial space.*

Proof Consider an FDS $\mathscr{D} = \langle \mathscr{V}, \theta, \rho\rangle$ and a CTL* formula φ. For every state subformula ψ of φ we compute the set of states $[\![\mathscr{D}, \psi]\!]$. As before, we assume an efficient way to represent, manipulate, and compare assertions.

For a proposition p and Boolean combinations of state formulas the treatment is no different from the treatment of CTL.

For $\psi = \mathbf{A}\psi_1$, where ψ_1 is an LTL formula we effectively apply the LTL model-checking algorithm for $\neg\psi_1$. Let $\mathscr{T}_{\neg\psi_1}$ be the temporal tester constructed for $\neg\psi_1$ as in Sect. 2.3 and let $\mathscr{X}$ be the set of variables of $\mathscr{T}_{\neg\psi_1}$. Consider the system $\mathscr{D} \,|||\, \mathscr{T}_{\neg\psi_1}$. Consider Algorithm 3 from Sect. 2.2. Recall that the system $\mathscr{D}$ is fairness-free. Thus, the only fairness requirements in the composition $\mathscr{D} \,|||\, \mathscr{T}_{\neg\psi_1}$ are the justice requirements of $\mathscr{T}_{\neg\psi_1}$. The algorithm computes the set of viable states of $\mathscr{D} \,|||\, \mathscr{T}_{\neg\psi_1}$. Consider a viable state (s, d) of $\mathscr{D} \,|||\, \mathscr{T}_{\neg\psi_1}$, where $s \in \Sigma_{\mathscr{V}}$ and $d \in \Sigma_{\mathscr{X}}$.

If $d \models x_{\neg\psi_1}$, then there is a fair path that starts in (s, d). In particular, the projections of this fair path on the variables in $\mathscr{V}$ and $\mathscr{X}$, respectively, show that there is a path starting in s that does not satisfy ψ_1. Thus, state s of $\mathscr{D}$ cannot satisfy $\mathbf{A}\psi_1$. Dually, for a state s that satisfies $\mathbf{A}\psi_1$ in $\mathscr{D}$, there cannot be a state d such that $d \models x_{\neg\psi_1}$ and (s, d) is viable. Let viable be the assertion characterizing the set of viable states of $\mathscr{D} \,|||\, \mathscr{T}_{\neg\psi_1}$. Then, $\mathsf{sat} \equiv \forall\mathscr{X}.\mathsf{viable} \rightarrow \neg x_{\neg\psi_1}$ is the assertion characterizing the set of states of $\mathscr{D}$ that satisfy $\mathbf{A}\psi_1$.

We now construct a new system $\mathscr{D}_1$ that embeds this information with a new Boolean variable x_ψ. Let $\mathscr{D}_1 = \langle \mathscr{V} \cup \{x_\psi\}, \theta_1, \rho_1 \rangle$, where $\theta_1 \equiv \theta \wedge (x_\psi \leftrightarrow \mathsf{sat})$ and $\rho_1 \equiv \rho \wedge (x'_\psi \leftrightarrow \mathsf{prime}(\mathsf{sat}))$. Furthermore, for every subformula $\tilde{\psi}$ of φ that contains ψ, we replace every occurrence of $\mathbf{A}\psi_1$ in $\tilde{\psi}$ by the new proposition x_ψ.

The treatment of formulas of the form $\mathbf{E}\psi_1$ is similar. We construct a temporal tester for ψ_1 and compute the set of viable states of $\mathscr{D} \,|||\, \mathscr{T}_{\psi_1}$. Then, we compute the assertion viable characterizing the viable states of $\mathscr{D} \,|||\, \mathscr{T}_{\psi_1}$. The assertion $\mathsf{sat} \equiv \exists\mathscr{X}.\mathsf{viable} \wedge x_{\psi_1}$ characterizes the states of $\mathscr{D}$ that satisfy $\mathbf{E}\psi$. We then construct the system $\mathscr{D}_1$ with the additional variable x_ψ and replace every occurrence of $\mathbf{E}\psi_1$ by x_ψ. □

As in the case of CTL, satisfiability is more complicated than model checking. Here, the techniques required to prove satisfiability involve the use of automata over infinite trees and determinization of automata over infinite words (cf. first proof [21]). We do not cover these techniques here and state the following.

Definition 16 (Satisfiability) A CTL* formula φ is satisfiable if some FDS implements it. That is, φ is satisfiable if there is an FDS $\mathscr{D}$ such that $\mathscr{D} \models \varphi$.

Theorem 16 *CTL* satisfiability is decidable in doubly exponential time.*

The reader interested in the proof is referred to [33] for the construction of automata on infinite trees that accept the set of models of CTL* formulas. The algorithm that then checks whether such an automaton accepts a non-empty language, corresponding to satisfiability of the original CTL* formula, is available, for example, in [52].

Acknowledgements We would like to thank the referees for insightful comments and spotting an error in an earlier version of the chapter, and B. Cook and H. Khlaaf for discussions on CTL.

References

1. Alpern, B., Schneider, F.B.: Recognizing safety and liveness. Distrib. Comput. **2**(3), 117–126 (1987)
2. Alur, R., Bouajjani, A., Esparza, J.: Model checking procedural programs. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)

3. Barrett, C.W., Tinelli, C.: Satisfiability modulo theories. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
4. Barringer, H.: Up and down the temporal way. Comput. J. **30**(2), 134–148 (1987)
5. Ben-Ari, M., Pnueli, A., Manna, Z.: The temporal logic of branching time. Acta Inform. **20**, 207–226 (1983)
6. Bloem, R., Chatterjee, K., Jobstmann, B.: Graph games and reactive synthesis. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
7. Bojanczyk, M.: The common fragment of CTL and LTL needs existential modalities. In: Amadio, R.M. (ed.) Intl. Conf. on Foundations of Software Science and Computational Structures (FoSSaCS). LNCS, vol. 4962, pp. 172–185. Springer, Heidelberg (2008)
8. Bouyer, P., Fahrenberg, U., Larsen, K.G., Markey, N., Ouaknine, J., Worrell, J.: Model checking real-time systems. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
9. Bryant, R.E.: Binary decision diagrams. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
10. Carmo, J., Sernadas, A.: Branching versus linear logics yet again. Form. Asp. Comput. **2**(1), 24–59 (1990)
11. Chandy, K.M., Misra, J.: Parallel Program Design: A Foundation. Addison-Wesley, Boston (1988)
12. Clarke, E.M., Draghicescu, I.A.: Expressibility results for linear-time and branching-time logics. In: de Bakker, G.R.J.W., de Roever, W.P. (eds.) Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, School/Workshop. LNCS, vol. 354, pp. 428–437. Springer, Heidelberg (1989)
13. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. Trans. Program. Lang. Syst. **8**(2), 244–263 (1986)
14. Doyen, L., Frehse, G., Pappas, G.J., Platzer, A.: Verification of hybrid systems. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
15. Eisner, C., Fisman, D.: Functional specification of hardware via temporal logic. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
16. Emerson, E.A.: Temporal and modal logic. In: van Leeuven, J. (ed.) Handbook of Theoretical Computer Science, vol. B: Formal Models and Semantics (B), pp. 995–1072. Elsevier/MIT Press, Amsterdam/Cambridge (1990)
17. Emerson, E.A., Clarke, E.M.: Using branching time temporal logic to synthesize synchronization skeletons. Sci. Comput. Program. **2**(3), 241–266 (1982)
18. Emerson, E.A., Halpern, J.Y.: Decision procedures and expressiveness in the temporal logic of branching time. J. Comput. Syst. Sci. **30**(1), 1–24 (1985)
19. Emerson, E.A., Halpern, J.Y.: "Sometimes" and "not never" revisited: on branching versus linear time temporal logic. J. ACM **33**(1), 151–178 (1986)
20. Emerson, E.A., Lei, C.L.: Modalities for model checking: branching time strikes back. In: Symp. on Principles of Programming Languages (POPL), pp. 84–96. ACM, New York (1985)
21. Emerson, E.A., Sistla, A.P.: Deciding full branching time logic. Inf. Control **61**(3), 175–201 (1984)
22. Gabbay, D.M., Pnueli, A., Shelah, S., Stavi, J.: On the temporal basis of fairness. In: Symp. on Principles of Programming Languages (POPL), pp. 163–173. ACM, New York (1980)
23. Harel, D., Pnueli, A.: On the development of reactive systems. In: Apt, K.R. (ed.) Logics and Models of Concurrent Systems. NATO ASI Series, vol. 13, pp. 477–498. Springer, Heidelberg (1985)
24. Holzmann, G.J.: Explicit-state model checking. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)

25. Jhala, R., Podelski, A., Rybalchenko, A.: Predicate abstraction for program verification. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
26. Kamp, J.: Tense logic and the theory of order. Ph.D. thesis, University of California, Los Angeles (1968)
27. Kesten, Y., Pnueli, A., Raviv, L.O., Shahar, E.: Model checking with strong fairness. Form. Methods Syst. Des. **28**(1), 57–84 (2006)
28. Kripke, S.: A completeness theorem in modal logic. J. Symb. Log. **24**(1), 1–14 (1959)
29. Kröger, F., Merz, S.: Temporal Logic and State Systems. Texts in Theoretical Computer Science. Springer, Heidelberg (2008)
30. Kupferman, O.: Automata theory and model checking. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
31. Kupferman, O., Pnueli, A.: Once and for all. In: Symp. on Logic in Computer Science (LICS), pp. 25–35 (1995)
32. Kupferman, O., Vardi, M.Y.: On bounded specifications. In: Nieuwenhuis, R., Voronkov, A. (eds.) Intl. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR). LNCS, vol. 2250, pp. 24–38. Springer, Heidelberg (2001)
33. Kupferman, O., Vardi, M.Y., Wolper, P.: An automata-theoretic approach to branching-time model checking. J. ACM **47**(2), 312–360 (2000)
34. Laroussinie, F., Markey, N., Schnoebelen, P.: Temporal logic with forgettable past. In: Symp. on Logic in Computer Science, vol. LICS, pp. 383–392. IEEE, Piscataway (2002)
35. Laroussinie, F., Schnoebelen, P.: A hierarchy of temporal logics with past. Theor. Comput. Sci. **148**(2), 303–324 (1995)
36. Laroussinie, F., Schnoebelen, P.: Specification in CTL+past for verification in CTL. Inf. Comput. **156**(1–2), 236–263 (2000)
37. Lichtenstein, O., Pnueli, A., Zuck, L.D.: The glory of the past. In: Logic of Programs, pp. 196–218 (1985)
38. Maidl, M.: The common fragment of CTL and LTL. In: Annual Symp. on Foundations of Computer Science (FOCS), pp. 643–652. IEEE, Piscataway (2000)
39. Manna, Z., Pnueli, A.: A hierarchy of temporal properties. In: ACM Symposium on Principles of Distributed Computing, pp. 377–410 (1990)
40. Peled, D., Vardi, M.Y., Yannakakis, M.: Black box checking. J. Autom. Lang. Comb. **7**(2), 225–246 (2002)
41. Peterson, G.L.: Myths about the mutual exclusion problem. Inf. Process. Lett. **12**(3), 115–116 (1981)
42. Pnueli, A.: The temporal logic of programs. In: Annual Symp. on Foundations of Computer Science (FOCS), pp. 46–57 (1977)
43. Pnueli, A., Zaks, A.: On the merits of temporal testers. In: Grumberg, O., Veith, H. (eds.) 25 Years of Model Checking. LNCS, vol. 5000, pp. 172–195. Springer, Heidelberg (2008)
44. Prior, A.: Time and Modality. Oxford University Press, Oxford (1957)
45. Queille, J.P., Sifakis, J.: Specification and verification of concurrent systems in CESAR. In: Dezani-Ciancaglini, M., Montanari, U. (eds.) Symp. on Programming. LNCS, vol. 137, pp. 337–351. Springer, Heidelberg (1982)
46. Rabin, M.: Weakly definable relations and special automata. In: Bar-Hillel, Y. (ed.) Proc. Symp. Math. Logic and Foundations of Set Theory, pp. 1–23. North-Holland, Amsterdam (1970)
47. Seshia, S.A., Sharygina, N., Tripakis, S.: Modeling for verification. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
48. Shankar, N.: Combining model checking and deduction. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
49. Sistla, A.P.: On characterization of safety and liveness properties in temporal logic. In: Malcolm, M.A., Strong, H.R. (eds.) ACM Symposium on Principles of Distributed Computing, pp. 39–48. ACM, New York (1985)

50. Sistla, A.P., Clarke, E.M.: The complexity of propositional linear temporal logics. J. ACM **32**(3), 733–749 (1985)
51. Sistla, A.P., Vardi, M.Y., Wolper, P.: The complementation problem for Büchi automata with applications to temporal logic. Theor. Comput. Sci. **49**, 217–237 (1987)
52. Vardi, M.Y.: Reasoning about the past with two-way automata. In: Larsen, K.G., Skyum, S., Winskel, G. (eds.) Intl. Colloq. on Automata, Languages, and Programming (ICALP). LNCS, vol. 1443, pp. 628–641. Springer, Heidelberg (1998)
53. Vardi, M.Y.: Branching vs. linear time: final showdown. In: Margaria, T., Yi, W. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 2031, pp. 1–22. Springer, Heidelberg (2001)
54. Vardi, M.Y., Wolper, P.: Reasoning about infinite computations. Inf. Comput. **115**(1), 1–37 (1994)
55. Wolper, P.: Temporal logic can be more expressive. Inf. Control **56**(1–2), 72–99 (1983)

Chapter 3
Modeling for Verification

Sanjit A. Seshia, Natasha Sharygina, and Stavros Tripakis

Abstract System modeling is the initial, and often crucial, step in verification. The right choice of model and modeling language is important for both designers and users of verification tools. This chapter aims to provide a guide to system modeling in four stages. First, it provides an overview of the main issues one must consider in modeling systems for verification. These issues involve both the selection or design of a modeling language and the steps of model creation. Next, it introduces a simple modeling language, SML, for illustrating the issues involved in selecting or designing a modeling language. SML uses an abstract state machine formalism that captures key features of widely-used languages based on transition system representations. We introduce the simple modeling language to simplify the connection between languages used by practitioners (such as Verilog, Simulink, or C) and various underlying formalisms (e.g., automata or Kripke structures) used in model checking. Third, the chapter demonstrates key steps in model creation using SML with illustrative examples. Finally, the presented modeling language SML is mapped to standard formalisms such as Kripke structures.

3.1 Introduction

A *model*, broadly speaking, can be thought of as the description of a system "on paper", or as a "virtual" system. This is in contrast to a "real" system, which can be thought of as a physical artifact: a car, a medical device, a Java program, or a stock market. Characterizing Java programs or stock markets as "physical" may seem strange; however, philosophical considerations aside, most people would agree that

S.A. Seshia (✉) · S. Tripakis
University of California, Berkeley, Berkeley, CA, USA
e-mail: sseshia@eecs.berkeley.edu

N. Sharygina
Università della Svizzera italiana, Lugano, Switzerland

S. Tripakis
Aalto University, Espoo, Finland

E.M. Clarke et al. (eds.), *Handbook of Model Checking*,
DOI 10.1007/978-3-319-10575-8_3

these systems are concrete and real enough to affect our lives in a direct way. Models also affect our lives, but in a more indirect way, as we discuss below.

The type of model is defined by its purpose. Models of existing systems are often referred to simply as "models", whereas models of systems yet to be built may be termed "specifications" or "designs". Some models are written using informal notations that are open to interpretation, whereas others are written in languages with mathematical semantics and are called "formal models". Most importantly, models rarely capture an entire, complete system, since the sheer size and complexity of most systems make this an impossible task. As a result, models usually focus only on "relevant" parts of a system and/or only on specific aspects of a system. For example, a model of a system that involves both hardware (HW) and software (SW) may only focus on the SW part; or a model may only focus on the logical aspects of a communication system, e.g., the communication protocol, and ignore other aspects such as performance (e.g., throughput, latency) or energy consumption.

Models are essential for our lives. They likely always have been. Humans as well as many other organisms need to form in their brain internal representations of how the external world "works". These representations can be seen as models. Closer to the focus of this book, engineering and technology heavily rely on mathematical models. In fact, the tasks of designing and building a system are intimately linked with various modeling tasks. Specification models are used to communicate the goals and requirements of a system among different engineering teams. Detailed design models are built in order to estimate behavior of a system prior to its construction. This is essential in order to avoid the costs and dangers of building deficient systems. After a system is built, models are still essential in order to operate and maintain the system, calibrate and tune it, monitor abnormal behavior, and ultimately upgrade it.

The goal of this chapter is to illuminate the key issues in modeling systems for formal verification, in general, and model checking, in particular. Because there are so many different types of systems and application domains, with varying concerns, even within the field of formal verification there is a plethora of modeling languages, formalisms, and tools, as well as modeling techniques, uses, and methodologies. It is beyond the scope of this chapter to give a thorough account of these; such an account would probably require an entire book by itself. Rather, this chapter has three more modest aims. First, we seek to provide an overview of the main issues one must consider in modeling systems for verification. These issues involve both the selection or design of a modeling language and the steps of model creation. Second, we introduce a simple modeling language, SML, to illustrate the issues involved in selecting or designing a modeling language. SML uses an abstract state machine formalism that captures key features of widely used languages based on transition system representations, and can serve to bind together the modeling languages introduced or used throughout the Handbook. By introducing SML, we seek also to simplify the connection between real languages (say Verilog, Simulink, or C) used by practitioners and various underlying formalisms (e.g., automata or Kripke structures) used in model checking. Finally, the chapter demonstrates key steps in model creation using SML with illustrative examples drawn from three different domains: hardware, software, and cyberphysical systems.

There are at least two possible audiences for the material in this chapter. The first are *users* of verification tools who need to make decisions about the right modeling language and verification tool for their task, and how best to model the system so as to get useful results from the chosen tool. The second audience comprises *tool builders* who may want to choose the appropriate modeling constructs for their domain (e.g., a tool for synthetic biology) or for easing the verification of a particular class of problems (e.g., parameterized systems with large data structures). Researchers working in related areas, such as program synthesis, may also find the concepts discussed in this chapter useful for better understanding the characteristics of verification techniques that they build upon.

We begin this chapter, in Sect. 3.2, by discussing the main issues one must consider in modeling systems for verification. In Sect. 3.3, we present SML, a simple language that illustrates many of the aspects of formal modeling languages. Three illustrative examples of system modeling with SML are presented in Sect. 3.4. We relate SML to the well-known formalism of Kripke structures in Sect. 3.5, and conclude in Sect. 3.6.

3.2 Major Considerations in System Modeling

Models are built for different reasons. It is important to emphasize that models are primarily tools used to achieve a certain purpose. They are means to a goal, rather than the goal itself. As such, the notion of a model being "good" or "bad" has little meaning by itself. It is more appropriate to examine whether a model is good or not *with respect to a certain goal*. For example, a model may be good for estimating the throughput of a system but useless for checking whether the system has deadlocks, or vice versa.

While models are built with many different goals in mind, they generally support the system design process. Stakeholders in this process must choose the right modeling formalisms, languages, and tools, to achieve their respective goals. As in [17], we distinguish between a modeling *formalism* and a modeling *language*. Formalisms are mathematical objects consisting of an abstract syntax and a formal semantics. Languages are concrete implementations of formalisms. A language has a concrete syntax, may deviate from the formalism in the semantics that it implements, and may implement multiple semantics (e.g., changing the type of the numerical solver in a simulation tool may change the behavior of a model). Also, a language may implement more than one formalism. Finally, a language usually comes together with a tool such as a compiler, simulator, or model checker. As an example of the distinction between formalisms and languages, timed automata [4] is a formalism, whereas Uppaal timed automata [47] and Kronos timed automata [31] are languages.

In this section, we discuss some of the main factors one must consider while choosing a modeling formalism and the challenges in modeling. We also give a brief survey of some modeling languages used for model checking.

3.2.1 Selecting a Modeling Formalism and Language

Here are some of the main factors one typically considers when selecting a good modeling formalism and language, for formal verification in general and model checking in particular:

- type of system;
- type of properties;
- relevant information about the environment;
- level of abstraction;
- clarity and modularity;
- form of composition;
- computational engines; and
- practical ease of modeling and expressiveness.

We discuss each of these factors in more detail below.

3.2.1.1 Type of System

Different modeling formalisms have been developed based on the character of the system being modeled. Some of the more common formalisms include:

- for *discrete(-time)* systems, formalisms such as finite state machines and pushdown automata [38, 44], extended state machines with discrete variables, hierarchical extensions such as Statecharts [35], as well as more declarative formalisms such as propositional temporal logics [52];
- for *continuous(-time)* systems, formalisms such as ordinary differential equations (ODEs) and differential algebraic equations (DAEs);
- for *concurrent processes*, formalisms such as communicating sequential processes (CSP) [36], a calculus of communicating systems (CCS) [53], the pi-calculus [54], Petri nets [56], marked graphs [26, 43], etc.;
- for *compositional modeling*, formalisms such as process algebras [33, 36, 53], and Reactive Modules [6];
- for *dataflow* systems, formalisms such as Kahn process networks [42] and various subclasses such as synchronous dataflow (SDF) [48], Boolean dataflow (BDF) [21], scenario-aware dataflow (SADF) [59], etc.;
- *scenario-oriented* formalisms such as message sequence charts [40, 41] and live sequence charts [28];
- for *timed* and *hybrid* systems, which combine discrete and continuous dynamics, formalisms such as timed and hybrid automata [3, 4], or real-time temporal logics [2, 5];
- *discrete-event formalisms* for timed systems, such as the denotational ones proposed in [9, 18, 50, 62], as well as operational ones inspired by discrete-event simulation and tools like Ptolemy [58, 61];

- *probabilistic* variants of many of the above formalisms, such as Markov chains, Markov Decision Processes, stochastic timed and hybrid automata, etc., for example, see [7, 8, 30, 39, 45];
- *game-* or *cost-theoretic* variants of some of the above formalisms, focusing on optimization and synthesis, instead of analysis, for example, see [12, 23, 24].

In addition to the above formalisms which focus on somewhat specific classes of systems, the need for modeling *heterogeneity*, that is, capturing systems that combine semantically heterogeneous components, such as timed and untimed, discrete and continuous, etc., has resulted in heterogeneous modeling frameworks such as Focus [18], Ptolemy [32], or Metropolis [10, 29], and corresponding formalisms [13, 18, 61].

3.2.1.2 Type of Property

During verification, the system model is coupled with a specification of the property to be verified. Several classes of properties exist, each supported by corresponding specification languages. Examples of specification languages for reactive systems include computation tree logic, regular expressions, Statecharts diagrams, graphical interval logics, a modal mu-calculus and a linear-time temporal logic just to name a few (more details can be found in Chap. 2 on temporal logic).

The choice of specification language and system model usually depends on the type of property one wishes to verify. For example, if one wishes to verify real-time properties of a system's execution over time, a real-time temporal logic might be the right choice of specification, and the system might best be represented as a timed automaton or timed CSP program. On the other hand, if for the same system one only wishes to verify Boolean properties such as absence of deadlock, then propositional temporal logic might suffice as the specification language.

It should be noted that the property specification language is not necessarily separate from the language used to model the system under verification. Often, the latter language provides mechanisms such as *assertions* or *monitors* that can be used to specify (usually safety) properties.

3.2.1.3 Modeling the Environment

One of the trickiest aspects of verification, which can both miss bugs and create spurious ones, is the task of modeling the environment of the system. The environment is usually much larger than the system, essentially incorporating everything other than the system under verification. On the other hand, it is also likely to be the part of the system that is least well understood, since often even a complete description of the environment is not available.

For example, in software model checking, one often needs to model the libraries that a piece of code uses, which form its environment. If only propositional temporal properties involving the sequence of system calls are relevant, then environment

models such as finite automata representing the language of system calls generated by a library component might be sufficient.

3.2.1.4 Level of Abstraction

The *level of abstraction*, i.e., the detail or faithfulness of a model, is an essential consideration in modeling. A highly detailed model may be hard or impossible to build due to time and cost constraints. Even if it can be built, it may be too large or complex for it to be amenable to (manual or automated) analysis. *State explosion* is a well-known phenomenon that plagues many techniques such as model checking. *Abstraction* methods are essential in building simpler and smaller models, by hiding unnecessary details. The difficulty is in understanding which details are truly irrelevant. Errors in this task often result in models that omit critical information. This may compromise the faithfulness of a model and ultimately render it useless.

As an example of successful use of abstraction, consider the process of modeling cache coherence protocols. Typically, one models the messages sent by the various processors as belonging to an abstract enumerated data type rather than represent the specific data formats actually used in the implementation of the protocol. Such abstraction is usually appropriate for the verification task, which depends only on the sequence and type of messages sent, rather than the specific bit-encoding of the message formats.

Verifiers that build models automatically from code usually rely heavily on automatic abstraction in the process. Selecting the right modeling formalism for such tools is usually critical to effective abstraction. Chapters 10 and 13 in this Handbook provide more detail on techniques for automatic abstraction.

3.2.1.5 Clarity and Modularity

Models are often, although not always, designed to be viewed by humans. In such cases, the models must be clear and easy to understand. One way to ensure this is to use a modular approach in constructing the model. Typically a modeling language will include some notion of a *module* or *process*, and provide means to combine or compose such modules/processes into larger entities. Such notation usually also includes ways to hide internal details during the composition process, so as to retain only essential information in the interfaces of modules [60]. In addition to making models easier to understand, modularity can sometimes also be exploited to make the verification task itself easier, e.g., by using compositional techniques (see Chap. 12 on compositional methods).

A related point is the specific form of composition of modules, which we discuss next.

3.2.1.6 Form of Composition

Systems are rarely constructed monolithically. They are usually built by combining and modifying existing *components*. The form in which such components interact can determine the modeling formalism that is suitable for verification.

For example, consider a sequential circuit built up by connecting several modules, all of which share the same clock. Since all modules step on the same clock tick, a *synchronous* composition of these modules is a suitable choice, even when many of these modules are represented at a very coarse level of abstraction.

Similarly, consider modeling a distributed database that uses a protocol to ensure that replicated state is consistent. Different nodes connected through the Internet are unlikely to share a synchronized clock, and hence, for this problem, *asynchronous* composition is the appropriate form of composition.

For some systems, a hybrid of synchronous and asynchronous composition might be suitable; for example, processes might synchronize on certain input actions while stepping asynchronously otherwise. This is particularly the case in formalisms such as timed and hybrid automata, where processes synchronize in time (i.e., time elapses at the same rate for all processes) while their discrete actions may be asynchronous.

Notions such as synchronous/asynchronous composition are particularly relevant in formalisms with operational semantics, such as transition systems. In formalisms with denotational semantics, other forms of composition may be better suited. For instance, in Kahn Process Networks [42] processes are typically viewed as functions from streams to streams, composition is defined as functional composition, and fixpoint theory is used to give semantics to feedback. In continuous-time formalisms processes may also be seen as functions manipulating continuous-time signals, and functional composition may be used here as well.

3.2.1.7 Computational Engines

A final consideration for modeling is the availability of suitable, scalable computational engines that power the verification tools for that class of models. For finite-state model checking, these include Binary Decision Diagrams (BDDs) [19] and Boolean satisfiability (SAT) solvers [51]. For model-checking software and high-level models of hardware, satisfiability modulo theories (SMT) solvers [11] play a central role. (See also Chaps. 7, 9, and 11 in this Handbook for further details on BDDs, SAT, and SMT.)

Even within the realm of SAT and SMT solvers, modeling plays an important role in ensuring the scalability of verification. For example, hardware designs can be represented most naturally at the bit-vector level, where signals can take bit-vector values or be arranged into arrays (memories) of such bit-vector values. One strategy that has proved highly effective for control-dominated hardware designs is to "bit-blast" the model to generate a SAT problem whose solution determines the result of verification. However, for many data-dependent properties, or for proving

equivalence or refinement of systems, one might need a higher level of abstraction. It is here that techniques for automatically abstracting designs to a higher "term level" come in handy—functional blocks can be abstracted using uninterpreted or partially-interpreted functions, and data words can be represented as abstract terms without regard to their specific bit-encoding (see, for example, [16, 20]). Such a representation then enables the use of SMT solvers in a rich set of theories including linear and non-linear arithmetic over the integers and reals, arrays, lists, uninterpreted functions, and bit-vector arithmetic.

3.2.1.8 Practical Ease of Modeling and Expressiveness

Even though theoretically two modeling languages may be equivalent in terms of expressiveness, in practice one may be much easier to use than another. For example, a language which provides no explicit notion of variables but requires users to encode the values of variables in the control states of an automaton is cumbersome to use except for toy systems. Also, a language which only provides Boolean data types is harder to use than a language which offers bounded integers or user-defined enumerations, even though the two languages are theoretically equivalent in terms of expressiveness. As a final example, a language which allows declaration of process *types* and then creation of multiple process *instances*, each with different parameters, is easier to use than a language which requires every process instance to be created in the model "by hand".

3.2.2 Modeling Languages

As mentioned earlier, it is useful to distinguish between formalisms, which are mathematical objects, and concrete modeling languages (and tools) which support such formalisms. A plethora of modeling languages exist, developed for different purposes, including:

- *Hardware description languages (HDLs).* These languages have been developed for modeling digital, analog, or so-called *mixed-signal* (combining digital and analog) circuits. Verilog, VHDL, and SystemC are widespread HDLs. Tools implementing HDLs provide features such as simulation, formal verification in some cases, and most importantly, automatic implementation such as logic synthesis and layout.
- *General-purpose modeling languages,* such as UML and SysML. These languages aim to capture many different aspects of software and systems in general, and offer different sub-languages implementing various formalisms, from hierarchical state machines to sequence diagrams.
- *Architecture Description Languages (ADLs),* such as AADL. These languages aim to be system-level design languages, for software and other domain-specific systems (e.g., originally avionics systems, in the case of AADL).

- *Simulation-oriented languages and tools,* such as Matlab-Simulink or Modelica. These languages have their origin in modeling and simulation of physical systems and support ODE and DAE modeling. They have recently evolved, however, to encompass discrete models such as state machines, and to target the larger domain of control, embedded, and cyber-physical systems. Simulink and related tools provide primarily simulation, but also code generation and even formal verification in some cases.
- *Reactive programming languages,* such as the synchronous languages Lustre [22] and Esterel [14]. These languages were initially conceived as programming languages for reactive, real-time, and embedded systems. As a result, the tools that come with these languages are typically compilers and code generators, typically providing simulation for debugging purposes. However, synchronous languages and tools sometimes also provide exhaustive verification features, and include mechanisms for modeling environment assumptions and non-determinism. As a result, these languages can be used for more general modeling purposes, too.
- *Verification languages.* These languages have been developed specifically for formal verification purposes, using model-checking or theorem-proving techniques, including satisfiability solving. This class is the main focus of the present chapter.

It is beyond the scope of this chapter to provide a complete survey of modeling languages. As the topic of this book is formal verification, we focus on modeling languages developed specifically for verification purposes. However, even within this narrower domain, we can only list a small selection of languages, among all those proposed in the formal verification literature.

The list is presented in Table 1. Each language has been tailored to the particular kind of verification problem that it was designed to model, and the engines that underlie the corresponding tool. We have classified the languages along five dimensions: the supported formalism, data types, form of composition, properties (safety or liveness) and the underlying computational engines. This listing is not meant to be exhaustive; rather the goal is to give the reader a flavor of the range of languages used in model checking today. We also note that several languages listed in the table were inspired by other formalisms; for example, the SAL language listed in Table 1 was inspired, in part, by the Reactive Modules formalism [6]. Further, it is important to remember that even verification tools that operate "directly" on programming languages such as C or Verilog extract some sort of formal model first, and this formal representation is typically very similar to modeling languages such as those listed in Table 1.

3.2.3 Challenges in Modeling

Beyond the computational difficulties of analyzing the models (e.g., state explosion during model checking, trace/time explosion during simulation, etc.) there are other difficulties in modeling that may be viewed as being at a more high level, and thus harder to address. Some of these difficulties are as follows:

Table 1 A selection of verification languages and their main characteristics

Language	Supported formalisms	Data types	Composition	Properties	Engines
SMV	Discrete (finite-state systems)	Booleans, finite-domain	Synchronous (primarily)	Safety and liveness	BDDs, SAT
SPIN/Promela	Discrete (finite-state systems)	Enumerative, channels, records, ...	Asynchronous, message passing, rendezvous	Safety and liveness	Explicit-state, partial order reduction, bit-state hashing
Murphi	Discrete (finite-state systems)	Booleans, finite-domain	Asynchronous	Safety and liveness	Explicit-state
SAL	Discrete and hybrid systems (finite/infinite-state transition systems)	Booleans, integers, reals, bit-vectors, arrays, ...	Synchronous and asynchronous	Safety and liveness	BDDs, SAT, and SMT
Alloy	Discrete (relational models)	Booleans, integers, bit-vectors, relations	Both (declarative relational modeling)	Safety	SAT
UCLID	Discrete (finite/infinite-state transition systems)	Booleans, integers, bit-vectors, arrays, restricted lambdas, ...	Synchronous and asynchronous	Safety	BDDs, SAT, and SMT
Uppaal	Timed (timed automata)	Enumerative, arrays, records, ...	Asynchronous with rendezvous	Safety and liveness	Difference-bound matrices, polyhedral reasoning
SpaceEx	Hybrid (hybrid automata)	Reals, discrete events	Asynchronous with synchronization on labels	Safety	Abstraction, polyhedral reasoning
Spec#/Boogie	Discrete (sequential programs)	Integers, bit-vectors, arrays, and other data structures	Object-oriented modules	Safety	Theorem proving, SMT

- Except in cases where the model is generated automatically (e.g., extracted from code), modeling is a creative process which can be quite difficult to get right.
- The choice of a modeling language/formalism is currently more of an art than a systematic science. There are few guidelines on how to go about this, and often the choice is dictated by historical or other reasons (e.g., company tradition, legacy models, etc.).
- Even after a model is constructed, it can be difficult to know whether the model is good, complete, or consistent. For specifications, this boils down to the problem often stated as: "have we specified enough properties?"
- Since models can themselves be incorrect or inconsistent, one must carefully interpret the results of verification. For example, in model checking, when a model fails to satisfy a property, is the model of the system wrong, or is the property incorrect? Such analysis usually requires some human insight.
- Constructing models of the environment, in particular, can be extremely tedious and error-prone. For instance, in automatic synthesis of systems from specifications (e.g., temporal logic), the process of writing down the specification, including constraints on the environment, is usually one of the hardest tasks. Similarly, for the problem of timing analysis of embedded software, many techniques involve having a human engineer painstakingly construct an abstract timing model of a microprocessor for use in software timing analysis (e.g., see [57] for a longer discussion). Automating the construction of environment models is an important challenge for extending the reach of formal verification and synthesis.

3.2.4 Scope of This Chapter

In the rest of this chapter, we seek to give a flavor of the above issues by introducing a simple modeling language, SML, and using it to model a small but diverse collection of systems. SML adopts many of the common features of modeling languages such as the idea of modeling a system as a *transition system* or an *abstract state machine* [34]. SML may not be the best fit, or even expressive enough, to model all types of systems and properties. With this in mind, we have chosen to make SML parametric with respect to its data types and the operations on those data types, illustrating with examples how different types of systems can be captured in the SML syntax. Some chapters in this Handbook will introduce modeling formalisms of their own, which are similar to SML, but differ in their emphasis; for example, Chap. 16 on Software Verification introduces a transition system formalism that emphasizes aspects important in verifying safety properties of programs, such as having a special variable to model the program counter, and the notion of an error condition. Using SML we emphasize the issues that arise across various system types, such as modularity, types of composition, and abstraction levels.

Fig. 1 Module Syntax in SML. A primitive module does not have the `composition`, `sharedvars`, `instances`, and `connect` sections

```
module M :
        inputs :        i_1 : τ_1; i_2 : τ_2; ...; i_k : τ_k;
        outputs :       o_1 : τ'_1; o_2 : τ'_2; ...; o_m : τ'_m;
        statevars :     v_1 : τ''_1; v_2 : τ''_2; ...; v_l : τ''_l;
        sharedvars :    u_1 : τ'''_1; u_2 : τ'''_2; ...; u_h : τ'''_h;
        init :          α;
        trans :         δ;
        composition :   synchronous | asynchronous;
        instances :     M_1, M_2, ..., M_n;
        connect :       γ;
```

3.3 Modeling Basics

We define a language for modeling abstract state machines called SML, which stands for "Simple Modeling Language". In this section, we present the syntax and semantics of SML.

3.3.1 Syntax

An SML program is made up of *modules*. Syntactically, an SML program is a list of *module* definitions. Each module definition comprises a module name followed by a module body. A module body, in turn, is made up of a list of definitions:

- a list of *input variable declarations*, $i_1 : \tau_1, i_2 : \tau_2, \ldots, i_k : \tau_k$;
- a list of *output variable declarations*, $o_1 : \tau'_1, o_2 : \tau'_2, \ldots, o_m : \tau'_m$;
- a list of *state variable declarations*, $v_1 : \tau''_1, v_2 : \tau''_2, \ldots, v_l : \tau''_l$;
- a list of *shared variable declarations*, $u_1 : \tau'''_1, u_2 : \tau'''_2, \ldots, u_h : \tau'''_h$; and
- a *behavior definition* which defines the transition and output relations of the module.

Some state variables may also be output variables. Shared variables are used for communication between asynchronously composed modules.

Declarations of variables may be omitted when the corresponding list is empty, e.g., a module with no shared variables will have no `sharedvars` section. Each variable has an associated type (domain of possible values), indicated above by the τ_i variables. We leave the types unspecified in this section; see Sect. 3.4 for examples.

The syntax of a module is given in Fig. 1. It is useful to make a distinction between two kinds of modules:

- *primitive modules*, which are not made up of simpler modules, and thus omit the `composition`, `instances`, and `connect` sections of the syntax; and
- *composite modules*, which are compositions of simpler modules, and thus include these sections.

A behavior definition of a primitive module comprises an *initial state definition* followed by a *transition relation definition*.

An initial state definition is a formula on state variables. A transition relation definition is a formula on input, output, state, and "next-state" variables of the module. A next-state variable is of the form `next(`v`)` where v is a state variable.

A composite module, in addition to the initial state and transition relation definitions, also includes:

- a *composition-instances declaration*, which declares the module instances that comprise the composite module, and the form of composition, either *synchronous* or *asynchronous*; and
- a *connections definition*, which is a list (denoting conjunction) of binary equalities between two variables of the instances and of the composite module, or between a variable and a constant.

For a composite module, the transition relation section defines how variables defined locally in this module (i.e., not in sub-modules) evolve.

Given two instances `m1` and `m2` of a module M containing a variable `x`, we refer to the instances of `x` in `m1` and `m2` as `m1.x` and `m2.x`.

For examples of primitive modules, see the Constant, Scale, Difference, and DiscreteIntegrator modules in Fig. 11. For examples of composite modules, see the Helicopter and System modules in Fig. 11.

3.3.2 Dynamics

We give semantics to an SML program by viewing it as a *symbolic transition system* (STS). In Sect. 3.5, we will relate STSs to one of the classical modeling formalisms, Kripke structures.

An STS is a tuple $(I, O, V, U, \alpha, \delta)$ where:

- I, O, V, U are finite sets of input, output, state and shared variables, each variable also having an associated type;
- α is a formula over $V \cup U$ (the initial states predicate);
- δ is a formula over $I \cup O \cup V \cup U \cup V' \cup U'$ (the transition relation), where $V' = \{s' \mid s \in V\}$ is a set of primed state variables representing the "next-state variables", and similarly with U'.

Given an SML module M as in Fig. 1, we define the STS for it as follows. If M is primitive, its syntax directly defines the tuple $(I, O, V, \emptyset, \alpha, \delta)$ (the set of shared variables is empty). If M is composite, then we define its STS in terms of the STSs for its constituent module instances. Suppose M is a composition of module instances $M_1, M_2, \ldots, M_n$, with corresponding STSs of the form $(I_i, O_i, V_i, U_i, \alpha_i, \delta_i)$, for $i = 1, \ldots, n$. Let β be the predicate derived from the initial state declaration of M (not its constituent modules). Let γ be the predicate derived from the connections declarations of M as a conjunction of each of the

equations making up those declarations. Then, the STS for M is $(I, O, V, U, \alpha, \delta)$, where:

- I is $(\bigcup_{i=1}^{n} I_i) \setminus I_0$, where I_0 is the set of those input variables that are connected to an output variable in the connections declarations of M;
- O is $\bigcup_{i=1}^{n} O_i$;
- V is $\bigcup_{i=1}^{n} V_i$;
- U is $\bigcup_{i=1}^{n} U_i$;
- α is $\beta \wedge \bigwedge_{i=1}^{n} \alpha_i$;
- δ depends on the form of composition:
  - For synchronous composition, δ is defined as $\gamma \wedge \bigwedge_{i=1}^{n} \delta_i$.
  - For asynchronous composition there are a couple of choices. We will define here the choice of *interleaving semantics*, where a transition of the composition involves one module taking a transition while all others *stutter* with their state variables remaining unchanged. Under this choice, δ is $\gamma \wedge \bigvee_{i=1}^{n} (\delta_i \wedge \bigwedge_{j \neq i} \sigma_j)$, where $\sigma_j = \bigwedge_{v \in V_j} v = v'$ defines stuttering of module M_j.

3.3.3 Modeling Concepts

Open and Closed Systems. A closed system is one that has no inputs. Any system with one or more inputs is open. In verification, we typically deal with closed systems, obtained by composing the system under verification with (a model of) its environment. In SML, we typically model interacting open systems as individual SML modules, and their composition, which is to be verified, is a closed system.

Safety and Liveness. Modeling formalisms for verification must be designed for the class of properties to be verified. In this regard, the most general categorization of properties is into *safety* and *liveness*. Formally, a property ϕ is a *safety property* if, for any infinite trace (execution) of the system, it does not satisfy ϕ if and only if there exists a finite prefix of that trace that cannot be extended to an infinite trace satisfying ϕ. We say that ϕ is a *liveness property* if every finite-length execution trace can be extended to an infinite trace that satisfies ϕ.[1] Chapter 2 gives several examples of safety and liveness properties, expressed in temporal logic.

Models encode safety or liveness concerns in different ways. Safety properties are defined by the *transition relation* of the model. With suitable encoding, the violation of a safety condition can be viewed as taking one of a set of "bad" transitions. Thus, by allowing certain transitions and disallowing others, a model can restrict its permitted executions to those adhering to a safety property. On the other hand, liveness properties are usually represented using *fairness conditions* on infinite execution paths of a model. We discuss fairness in more depth below, but, in essence,

[1] See papers by Lamport [46] and Alpern and Schneider [1] for a detailed treatment of safety and liveness.

fairness constraints can be used to rule out ways in which a finite-length execution of the model can be extended to an infinite execution violating the desired liveness condition.

Fairness. An important concept in execution of composed modules is fairness. Fairness constraints block infinite executions that are not realistic for concurrent systems, and are often required to demonstrate liveness properties. In other words, the fairness constraints are needed to ensure a proper resolution of the nondeterministic decisions taken during the execution of a concurrent system: no module gets neglected and each module always makes progress.

There are two major types of fairness: *strong* and *weak*. Weak (Buchi) fairness establishes that a step in the module execution cannot be enabled *forever* without being taken. Strong (Streett) fairness guarantees that a step cannot be enabled *infinitely often* without being taken.

Fairness can be useful in many different verification settings. For instance, consider the asynchronous (interleaved) composition of modules (considered above). One usually requires (weak) fairness in this setting: no module should be enabled forever without proceeding. Operationally, this could be modeled using a Boolean variable that indicates whether or not a module is enabled to make a transition, and then specifying that this variable must be true infinitely often.

Definition 1 An execution of a composite system is *fair*, if all modules progress infinitely often.

Fairness can be useful for modeling behavior even within a single module. An example of this setting is presented in Sect. 3.4.1.5.

Encapsulation. The module interface is defined in terms of its inputs and outputs, i.e., the environment communicates with the module by updating its input variables, which in response reacts by updating its output variables. A good model will expose *all* internal state that can (should) be accessed by its environment and *only* that state: this is important for ensuring that the verifier does not miss bugs or generate spurious error reports.

Moore and Mealy machines. The SML notation can be easily used both for Mealy and Moore machines, the standard modeling formalisms. For Moore machines, the output relation simply takes the form $\bigwedge_{j=1}^{m} o_j = f_j(V)$, where f_j denotes the output function for output variable o_j. Similarly, for Mealy machines, the output relation would take the form $\bigwedge_{j=1}^{m} o_j = f_j(V, I)$.

3.4 Examples

In order to illustrate modeling with SML, we present three examples from three different problem domains: digital circuits (Sect. 3.4.1), control systems (Sect. 3.4.2), and concurrent software (Sect. 3.4.3). Each example is a simplified version of a design artifact arising in practice. In each case, we begin by presenting this design, its

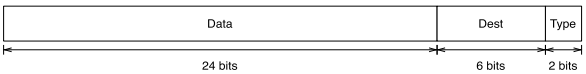


Fig. 2 Anatomy of a flit. A flit contains a 24-bit data payload, a 6-bit destination address, and a 2-bit type

simplification, and the corresponding SML model. We then describe how the SML model might need to be transformed in order to succeed at various verification tasks associated with the model. Our goal is to illustrate the various modeling considerations discussed in Sect. 3.2.1 in the particular context of each example.

3.4.1 Synchronous Circuits

Our representative example of digital circuits is a simple chip multiprocessor (CMP) router. First, we present a brief description of this example, and then describe it in SML notation.

3.4.1.1 Router Design

Network-on-chip (NoC) architectures are the backbone of modern, multicore processors and system-on-chip (SoC) designs, serving as the communication fabric between processor cores and other on-chip devices such as controllers for memory and input-output devices. It is important to prove that individual routers and networks of interconnected routers operate as per specification. The CMP router design [55] we focus on is part of an on-chip interconnection network that connects processor cores with memory and with each other. The main function of the router is to direct incoming packets to the correct output port. Each packet is made up of smaller components called *flits*. There are three kinds of flits: a *head flit*, which reserves an output channel, one or more *body flits*, which contain the data payload, and a *tail flit*, which signals the end of the packet. The typical data layout of a flit is depicted in Fig. 2. The two least significant bits represent the flit type, the next six bits represent the destination address, and the 24 most significant bits contain the data payload.

The CMP router consists of four main modules, as shown in Fig. 3. The *input controller* buffers incoming flits and interacts with the *arbiter*. Upon receipt of a head flit, the input controller requests access to an output port based on the destination address contained in the head flit. The arbiter grants access to the output ports in a fair manner, using a simple round-robin arbitration scheme. The remaining modules are the *encoder* and *crossbar*. When the arbiter grants access to a particular output port, a signal is sent to the input controller to release the flits from the buffers, and at the same time, an allocation signal is sent to the *encoder* which in turn configures the *crossbar* to route the flits to the appropriate output port.

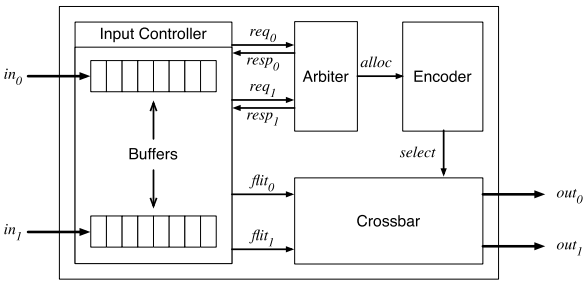


Fig. 3 Chip-Multiprocessor (CMP) Router. There are four main modules: the input controller, the arbiter, the encoder, and the crossbar

3.4.1.2 Simplifications and SML Model

Since the original router design is quite large, we make several simplifications to create a small example for this introductory chapter. First, we assume that each flit that makes up the packet raises a request to the arbiter, not just the head flit; this eliminates some logic tracking the type of flit. Second, we assume that the destination address, which indicates the output port to which a flit must be directed, is exactly two bits. Specifically, bits 2 and 3 of each flit encode the address, with `01` encoding output port 0 (bit 2 is `1` and bit 3 is `0`), and `10` encoding output port 1. The destination address is directly copied to the request lines of the arbiter; the encoding `00` for a request line indicates the absence of a request. Finally, we eliminate the Encoder module, directly using the `alloc` signals to route flits through the crossbar to the output ports of the router.

Figure 4 shows the SML representation of the router. We use the usual short form of declarations where variables of the same type are grouped together in the same declaration statement. The top-level module is termed `System`. Note that it has two inputs, and therefore is an open system. The top-level module is a synchronous composition of three modules: the input controller, the arbiter, and the crossbar. The input controller module in turn is a synchronous composition of two instances of the module modeling a FIFO buffer, plus some additional control logic to request access to an output port from the arbiter. Note how the buffers are defined as arrays mapping bitvectors to bitvectors, and implemented as circular queues. The arbiter uses a single priority bit to mediate access to an output port when both input ports request that same output port; if there is no conflict between output port requests, both can be granted simultaneously. The alloc signals generated by the arbiter are used in the crossbar to direct the flit at the head of the input buffers to the corresponding output port, if one is granted. If no flit is directed to an output port, the value of that output is set to a default invalid value `NAF` (standing for "not a flit").

3.4.1.3 Verification Task: Progress Through the Router

Consider the following verification problem stated in English below:

> Any incoming flit on an input port is routed to its correct output port, as specified by its destination address, within L clock cycles.

```
module InputController {
  inputs:  in0, in1 : bitvec[32];
           resp0, resp1 : bitvec[1];
  outputs: req0, req1 : bitvec[2];
           flit0, flit1 : bitvec[32];
  wires: deq0, deq1: bool;
  composition: synchronous
  instances:
     B0 : Buffer;  B1 : Buffer;
  connect:
     B0.flit_in = in0;
     B1.flit_in = in1;
     B0.dequeue = deq0;
     B1.dequeue = deq1;
     B0.flit_out = flit0;
     B1.flit_out = flit1;
  trans:
      deq0 = (resp0 != 0)
   && deq1 = (resp1 != 0)
   && req0 = flit0[3:2]
   && req1 = flit1[3:2];
}

module Buffer {
  inputs:  flit_in : bitvec[32];
           dequeue : bool;
  outputs: flit_out : bitvec[32];
  statevars:
      buf[SZ] : array bitvec[32] -> bitvec[32];
      num : bitvec[4];
      head : bitvec[4];
      tail : bitvec[4];
  init:
      num = 0x0 && head = 0x0 && tail = 0x0
      && for (i in 0x0..0x7):
           buf[i] = NAF;
  trans:
     flit_out = buf[head]
  && (flit_in != NAF) =>
         (num < SZ) =>
              next(tail) = (tail+1 mod SZ)
           && next(buf[tail]) = flit_in
           && next(num) = num+1
      && (dequeue && num > 0x0) =>
              next(head) = (head+1 mod SZ)
           && next(num) = num-1;
}
```

```
module Arbiter {
  inputs: req0, req1 : bitvec[2];
  outputs: resp0, resp1 : bitvec[1];
           alloc0, alloc1 : bitvec[2];
  statevars: priority : bool;
  init: priority = false;
  trans:
     req0 = 0 && req1 = 0 =>
       resp0 = 0 && resp1 = 0
  && req0 != 0 && req1 = 0 =>
       resp0 = 1 && resp1 = 0
  && req0 = 0 && req1 != 0 =>
       resp0 = 0 && resp1 = 1
  && req0 != 0 && req1 != 0 =>
        req0 = req1 =>
             priority =>
               resp0 = 0 && resp1 = 1
          && (!priority) =>
              resp0 = 1 && resp1 = 0
          && next(priority) = !priority
     && req0 != req1
          resp0 = 1 && resp1 = 1
  && (req0 = 01 && resp0 = 1) => alloc0 = 00
  && (req0 = 10 && resp0 = 1) => alloc0 = 01
  && (req1 = 01 && resp1 = 1) => alloc1 = 00
  && (req1 = 10 && resp1 = 1) => alloc1 = 01
  && (resp0 = 0 => alloc0 = 11)
  && (resp1 = 0 => alloc1 = 11);
}

module Crossbar {
  inputs: in0, in1 : bitvec[32];
          alloc0, alloc1 : bitvec[2];
  outputs: out0, out1 : bitvec[32];
  statevars: flit0, flit1 : bitvec[32];
  init: flit0 = NAF && flit1 = NAF;
  trans:
     out0 = flit0 && out1 = flit1
  && alloc0 = 00 => next(flit0) = in0
  && alloc0 = 01 => next(flit1) = in0
  && alloc1 = 00 => next(flit0) = in1
  && alloc1 = 01 => next(flit1) = in1
  && alloc0 = 11 & alloc1 = 00 => next(flit1) = NAF
  && alloc0 = 11 & alloc1 = 01 => next(flit0) = NAF
  && alloc1 = 11 & alloc0 = 00 => next(flit1) = NAF
  && alloc1 = 11 & alloc0 = 01 => next(flit0) = NAF;
}
```

```
module System {
  inputs: in0, in1 : bitvec[32];       outputs: out0, out1 : bitvec[32];
  composition: synchronous;
  instances: InpCtrl : InputController, Arb : Arbiter, Xbar : Crossbar,
  connect:
    in0 = InpCtrl.in0; in1 = InpCtrl.in1; out0 = Xbar.out0; out1 = Xbar.out1;
    InpCtrl.req0 = Arb.req0; InpCtrl.req1 = Arb.req1;
    InpCtrl.resp0 = Arb.resp0; InpCtrl.resp1 = Arb.resp1;
    Xbar.in0 = InpCtrl.flit0; Xbar.in1 = InpCtrl.flit1;
    Arb.alloc0 = Xbar.alloc0; Arb.alloc1 = Xbar.alloc1;
}
```

Fig. 4 A simplified chip multiprocessor router modeled in SML. `NAF` stands for "not a flit", and is implemented as the bitvector value `0x00000003`. We use a `wires` declaration to introduce new names for expressions. `SZ` is a parameter denoting the size of queues

This is a typical *latency bound property* that every router must satisfy. The bound L on the latency is left parametric for now.

A latency bound property with a fixed bound L falls within a broader class of properties known as *quality-of-service* properties.

It is common, however, to use an abstraction of this property that only requires L to be finite, but places no other requirement on its value. In essence, such a property specifies that any incoming flit is to be *eventually* routed to the correct output port.

Both forms of the latency bound property can be expressed in *temporal logic*, a specification formalism that is explored in greater depth in Chap. 2 on temporal logic.

```
module SimpleRouterEnv {
  inputs:  iflit0, iflit1 : bitvec[32]; // flits from router
  outputs: oflit0, oflit1 : bitvec[32]; // flits to router
  statevars:
      counter : bitvec[32];
      data0 : bitvec[28]; data1 : bitvec[28]; // changes non-determinisically
      dest0 : bitvec[2]; dest1 : bitvec[2]; // arbitrary value fixed for packet
  init:
      counter = 0x0 && (dest0 = 10 || dest0 = 01)
   && (dest1 = 10 || dest1 = 01) // non-deterministically assigned 01 or 10
  trans:
      next(counter) = counter + 1          // 32-bit finite-precision addition
   && next(dest0) = dest0 && next(dest1) = dest1 // stays unchanged for the packet
   && (counter = 0) =>
           oflit0 = (data0 @ dest0 @ 10) // head flit
        && oflit1 = (data1 @ dest1 @ 10) // head flit
   && (counter >= 1 || counter < PKT_SZ-1) =>
           oflit0 = (data0 @ dest0 @ 00) // body flit
        && oflit1 = (data1 @ dest1 @ 00) // body flit
   && (counter = PKT_SZ-1) =>
           oflit0 = (data0 @ dest0 @ 01) // tail flit
        && oflit1 = (data1 @ dest1 @ 01) // tail flit
   && counter >= PKT_SZ => oflit0 = NAF && oflit1 = NAF  // not a flit
}
```

Fig. 5 A simple environment model for the chip multiprocessor router modeled in Fig. 4. `PKT_SZ` is a parameter denoting the size of the packet (i.e., the number of flits per packet)

3.4.1.4 Data Type Abstraction

Consider a simple environment that injects exactly one packet into each input port, with the destination of the packets modeled by symbolic destination field in the respective head flit. Each packet comprises a head flit, several body flits, and a tail flit. By making the destination field symbolic, we can model both interesting scenarios: the case where the two injected packets are destined for different output ports as well as the case where they are headed for the same output port (resulting in contention to be resolved by the arbiter).

Figure 5 gives the SML code for the above environment model. Note that the outputs generated by this environment are the inputs to the top-level module `System` in the router design. These outputs are modeled as bit-vector variables.

The choice of how to model the data type of the flit can have a big impact on the scalability of verification. In work by Brady et al. [15], a router design almost identical to the one given in this chapter was considered for verification.[2] Two types of verification tasks were performed. In both cases, bounded model checking (BMC) (covered in Chap. 10) was used to check that starting from a reset state, the router correctly forwards both packets to their respective output ports within a fixed number of cycles that depends on the length of the packet (`PKT_SZ`). The difference was in how the data component of flits in all modules of the design were represented. In one case, the data component of each flit was modeled as a bit-vector 28 bits wide (as in Fig. 5), while in the second case this was modeled as an *abstract term*,

[2]The differences have to do with modeling the crossbar and routing logic more accurately than we have in this chapter, and are not significant for the discussion herein.

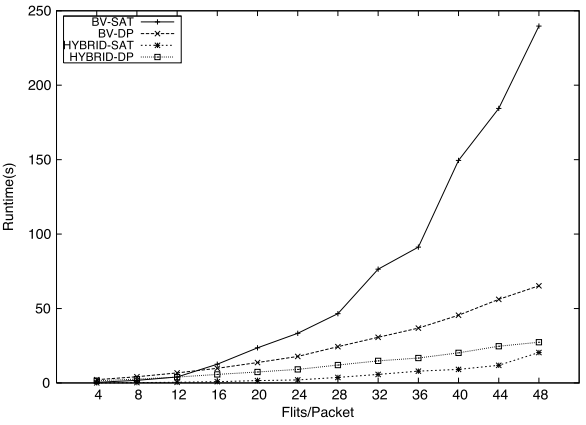


Fig. 6 Runtime comparison for increasing packet size in CMP router model. The runtimes for the bitvector-level and term-level designs are compared for increasing packet size. Runtimes for the underlying decision procedure are broken up into that required for generating a SAT solver encoding ("DP") and that taken for the SAT solving ("SAT")

which can be encoded with fewer bits by an underlying decision procedure for a combination of uninterpreted functions and equality and bit-vector arithmetic (see Chap. 11). In Fig. 6, we see that indeed the runtime of the verifier scales much better with increasing `PKT_SZ` in the case where such *term-level modeling* is employed than in the case where pure bitvector-level modeling is used.

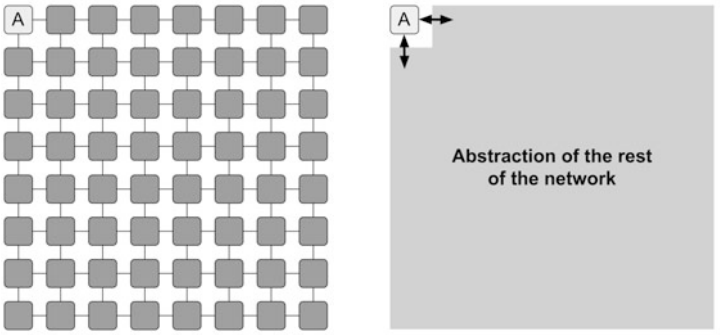
This example illustrates the points in Sect. 3.2.1 related to the right level of abstraction required to ensure that the underlying computational engine (a SAT-based decision procedure—also known as an SMT solver—in this case) can efficiently verify the problem at hand.

3.4.1.5 Environment Modeling

In the previous section, we considered the verification of a router with a rather contrived environment model that simply injects one packet into the router's input ports. In reality, such a router will be interconnected with other network elements in a specified topology. For example, consider an 8×8 grid of interconnected nodes shown in Fig. 7(a), where each node typically represents a router and network interface logic (e.g., connecting the router with a core or memory element). For this section, we will assume that each node is simply a router design similar to that given in Fig. 4, but with five input and output ports—four of these can be connected to neighboring nodes on the grid, and one can be connected to the processor or memory element associated with that node.

Next, consider a specific node in the grid labeled `A`. Suppose that we want to verify the property that every packet traveling through `A` spends no more than 15 cycles within `A`. One approach is to model the overall network as a synchronous composition of 64 routers, one for each node. However, this results in a model with tens of thousands of Boolean state variables, which is beyond the capacity of the best current formal verification tools.

An obvious alternative approach involves abstraction and modeling with nondeterminism. Specifically, as depicted in Fig. 7(b), one can abstract all nodes in the



(a) 8×8 network (b) Abstraction

Fig. 7 Verifying latency through a router in an on-chip network. We wish to verify a bound on the latency through node A

network other than A into an environment model E, where E is a transition system which, at each cycle, non-deterministically selects a port to send an input packet to A, sets the destination port for that packet, and chooses a data payload. If we apply this technique to the problem of verifying a latency bound of 15 through the router, it fails to prove the latency bound using a sequence of input packets that causes contention for the same output port within the router. The question is, however, whether such a pattern can arise in this particular network for the traffic patterns that the architect has in mind. Purely non-deterministic modeling of traffic sources does not permit us to model traffic sources in a more precise manner.

Thus, as mentioned in Sect. 3.2.1, the environment model must be tailored to the type of property to be verified. In this case, in order to prove a particular latency bound, we must come up with a suitable model of traffic sources (and sinks).

One formalism for more fine-grained modeling of any channel of NoC traffic (including sources and sinks) is a *bounded channel regulator*, introduced by Holcomb et al. [37] based on the regulator model described by Cruz [27]. A *channel* in an NoC is any link between components in the network. A bounded channel regulator $\mathcal{T}_{\mathcal{R}}$ is a monitor on a channel that checks three constraints: the *rate* ρ, *burstiness* σ, and bound B on the traffic that traverses the channel. Figure 8 illustrates a bounded channel regulator. The buffer of size σ begins filled with tokens, and one token is removed whenever a head flit passes through the channel being monitored. Src_{ρ} adds a token to the buffer once every ρ cycles, unless the buffer is already full. During a simulation, Src_{ρ} becomes inactive once it has produced a total of $B - \sigma$ tokens. If a head flit ever passes through the channel when the regulator queue is empty, then a Boolean flag a_i is set to signal that the channel traffic does not conform to the constraints of regulator i. We refer to a regulator with rate ρ, burstiness σ, and bound B as $\mathcal{T}_{\mathcal{R}}(\rho, \sigma, B)$. It is easy to model a bounded regulator in SML using non-deterministic assignment for generating packets, a FIFO buffer for storing tokens, and a counter for enforcing the rate of the traffic regulator.

The regulator can be applied to any channel in a network that is viewed as a generator of packets, such as a neighboring router, a processing element, or a memory

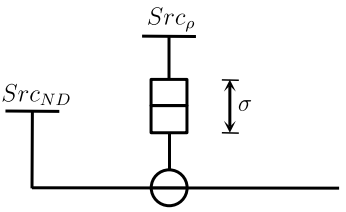


Fig. 8 Traffic regulator. The bounded channel regulator $\mathcal{T}_{\mathcal{R}}(\rho, \sigma, B)$ models all traffic patterns with average rate constrained by ρ, burstiness by σ, and total number of packets bounded by B

element. Such packet generators can be modeled as a completely non-deterministic source Src_{ND} (as shown in Fig. 8). Combining Src_{ND} with the regulator provides a way to restrict behaviors of the non-deterministic source that are overly adversarial compared to the actual design that this model abstracts away. It is also possible to model traffic sinks in a similar fashion.

Since router A in Fig. 7(b) is at a corner of the grid, it sees less incoming traffic than other routers in the center. With suitable parameters ρ, σ, and B for the bounded regulator model, one can verify the latency bound of 15 through the router A, as reported by Holcomb et al. [37]. The parameters can be generated manually or inferred automatically from traces generated from program executions.

3.4.1.6 Summary

In this section, we discussed the latency bound verification problem for a CMP router. Several of the considerations outlined in Sect. 3.2.1 arose. First, since the system was a digital circuit with a single clock, a discrete state machine formalism based on synchronous composition was suitable. Next, additional data type abstraction was necessary to reduce the search space for the underlying SAT-based computational engines. A suitable environment model had to be created to reason about a latency bound property. We started with a simplistic environment model, but then, in order to consider the more realistic whole-network scenario, had to formulate suitable models for sources (and sinks) of network traffic. This environment model also incorporates relevant abstractions to reduce the search space for verification.

3.4.2 Synchronous Control Systems

In the previous section we presented an example of a synchronous digital circuit modeled in SML. Synchronous digital circuits are an important class of systems that can be modeled using the synchronous model of computation. Another important class of systems which can often be viewed as synchronous systems are control systems, implemented either in hardware or software. Control systems typically consist of a *controller* which interacts with a *plant*, that is, a physical process to be controlled, in a closed-loop manner. The controller typically samples certain observable variables of the plant periodically through sensors, and issues commands to the plant through actuators.

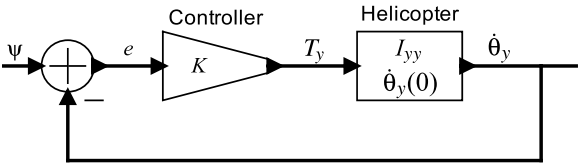


Fig. 9 A simple feedback control system. A proportional controller for a helicopter model (taken from [49])

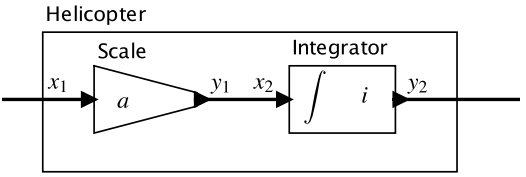


Fig. 10 A simple helicopter model. The Helicopter block of Fig. 9 (from [49])

In this section we present a toy feedback control system consisting of a simple proportional controller for a helicopter model. The example is an adapted version of the one described in Sect. 2.4 of [49]. There, the model is a *continuous-time* model. Here, we present a *discrete-time* (synchronous) version with the aim of illustrating how it could be captured in SML.

The system is shown graphically in a block-diagram notation in Fig. 9. It contains, at the top level, the Controller and Helicopter modules connected in feedback. The input of the Helicopter module is torque, denoted by T_y, and its output is angular velocity, which is the time derivative $\dot{\theta}_y$ of the angle θ_y. The input of the Controller is the error e, that is, the difference between the target angular velocity ψ (which we will take to be constant) and the actual angular velocity.

In this toy example the Controller is a simple proportional controller which multiplies the error by a constant K, that is, the controller implements $T_y = K \cdot e$. The Helicopter consists of two sub-modules, a Scale and an Integrator, as shown in Fig. 10.

The entire system, modeled in SML, is shown in Fig. 11. The top-level System module instantiates and connects four sub-modules, a Helicopter, a ProportionalController, a Constant (modeling the target angular velocity ψ) and a Difference module (computing the error e). The Helicopter module consists of a Scale and a DiscreteIntegrator (replacing the continuous integrator of Fig. 10). The ProportionalController consists simply of a Scale. The Constant, Scale, Difference and DiscreteIntegrator modules are primitive modules, while the rest are composite modules.

Properties of interest in control systems are stability, robustness, and control performance. Such properties can sometimes be expressed in formal specification languages such as temporal logic. For instance, in the helicopter example above, we may wish the error e to eventually become almost zero, and remain close to zero forever after. We may also want e to become almost zero within a certain deadline. Finally, we may also want e not to exceed certain upper and lower bounds as it converges to zero (i.e., bounded "overshoot" and "undershoot"). All these properties can be formalized in temporal logic.

```
module Constant {
  inputs:  value : real;
  outputs: out   : real;
  trans:
    out = value;
}

module Scale {
  inputs:    in    : real;
  outputs:   out   : real;
  statevars: coeff : real; // const
  trans:
    out = in * coeff;
}

module Difference {
  inputs:  in1 : real,
           in2 : real;
  outputs: out : real;
  trans:
    out = in1 - in2;
}

module DiscreteIntegrator {
  inputs:    in  : real;
  outputs:   out : real;
  statevars: sum : real;
     // initialized externally
  trans:
    out = sum &&
    next(sum) = in + sum;
}
```

```
module ProportionalController {
  inputs:    in    : real;
  outputs:   out   : real;
  statevars: coeff : real;
  composition: synchronous;
  instances: S : Scale;
  connect:
    S.coeff = coeff;
    S.in    = in;
    out     = S.out;
}


module Helicopter {
  inputs:    in    : real;
  outputs:   out   : real;
  statevars: coeff : real;
  init:
    I.sum = 0.0;
  composition: synchronous;
  instances: S : Scale,
             I : DiscreteIntegrator;
  connect:
    S.coeff = coeff;
    S.in    = in;
    I.in    = S.out;
    out     = I.out;
}
```

```
module System {
  composition: synchronous;
  instances: Heli : Helicopter,
             Ctrl : ProportionalController,
             Trgt : Constant,
             Diff : Difference;
  connect:
    Trgt.value = 0.0;
    Diff.in1   = Trgt.out;
    Diff.in2   = Heli.out;
    Ctrl.coeff = 10.0;
    Ctrl.in    = Diff.out;
    Heli.coeff = 10.0;
    Heli.in    = Ctrl.out;
}
```

Fig. 11 A toy synchronous control system modeled in SML

3.4.3 Concurrent Software

Concurrent software has long been a key application domain for model checking. In this section, we show how modeling a concurrent program for verification can require subtlety, even for small programs with limited concurrency.

```
    volatile uint timerCount = 0;
    void ISR(void) {
D ⟶    … disable interrupts
E ⟶    if(timerCount != 0) {
           timerCount--;
       }
       … enable interrupts
    }
    int main(void) {
       // initialization code
       SysTickIntRegister(&ISR);
       … // other init
A ⟶    timerCount = 2000;
B ⟶    while(timerCount != 0) {
        … code to run for 2 seconds
       }
    }
C ⟶ … whatever comes next
```

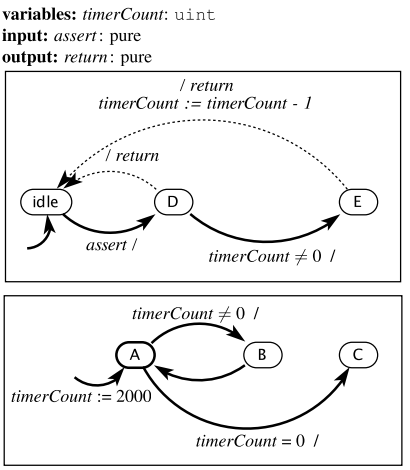


Fig. 12 Modeling a program that does something for two seconds and then continues to do something else (figure from [49])

Consider the program outlined in Fig. 12.[3] The main function seeks to perform some action (at the program location denoted B) for two seconds and then do something else (at location C). To keep track of time, the program uses a timer interrupt. The function `ISR` is registered as the interrupt service routine for this timer interrupt. The interrupt is raised every millisecond, so `ISR` will be invoked 2000 times in two seconds. The program seeks to track the number of invocations of `ISR` by using the shared variable `timerCount`.

Consider verifying the following property:

> The `main` function of the program will always reach position C.

In other words, will the program eventually move beyond whatever computation it was to perform for two seconds?

A natural approach to answer this question is to model both the `main` and `ISR` functions as state machines. In Fig. 12, we show two finite state machines that model `ISR` and `main`. The states of the FSMs correspond to positions in the execution labeled A through E, as shown in the program listing. Let us further assume that the `main` function loops in location C.

Note that these state machine models incorporate some important assumptions. One of these is on the atomicity of operations in the program. The program locations A through E are between C statements, so implicitly we are assuming that each C statement is an atomic operation—a questionable assumption in general. However, for simplicity, we will make this assumption here.

Another modeling assumption concerns the frequency with which interrupts can be raised. The raising of an interrupt is modeled in Fig. 12 using the input `assert`,

[3] This example is taken from a textbook on Embedded Systems [49].

which is a "pure" signal meaning that it is an event that is either present or absent at any time step (this can be modeled in SML as a Boolean). Assuming that interrupts can occur infinitely often, we can model the environment that raises interrupts as a state machine that non-deterministically raises an interrupt (sets `assert`) at each step.

The next question becomes how to compose these state machines to correctly model the interaction between the two pieces of sequential code in the procedures `ISR` and `main`. As first glance, one might think of *asynchronous composition* as a suitable choice. However, that choice is incorrect in this context because the interleavings are not arbitrary. In particular, `main` can be interrupted by `ISR`, but `ISR` cannot be interrupted by `main`. Asynchronous composition fails to capture this asymmetry.

Synchronous composition is also not a good fit. When `ISR` is running, the `main` function is not, and vice versa, unlike synchronous composition where the transition systems move in lock step.

Therefore, to accurately compose `main` and `ISR`, we need to combine them with a *scheduler* state machine. The scheduler has two modes: one in which `main` is executing, and one in which `ISR` is executing after it pre-empts `main`. This model also requires minor modifications to the state machines `main` and `ISR`. All three state machines are then composed together hierarchically and synchronously. Figure 13 shows the SML model for the combination.

The resulting machine is composed with its environment state machine, which models the raising of an interrupt. This final composition is asynchronous, reflecting the modeling assumption that the environment and the concurrent program do not share a common clock. However, we probably want to rule out the interleaving in which only the environment steps without ever giving the concurrent program a chance to execute. A fairness specification allows us to impose this modeling constraint.

With this model, we are able to verify that, in fact, the program does not satisfy the desired property. One counterexample involves an interrupt being repeatedly raised by `Env` and the program spends all its time in invocations of `ISR` with `main` unable to make any progress.

To summarize this example, we note that even the simplest concurrent programs, such as the interrupt-driven program of this section, can be tricky to model. Although conventional wisdom holds that asynchronous composition is the "right" choice for concurrent software verification, one must be careful to take into consideration relative priorities of tasks/processes and scheduling policies.

3.5 Kripke Structures

This section introduces Kripke Structures, perhaps the most common formalism for specifying system models. It then defines the mapping between the constructs of the modeling language defined in Sect. 3.3 and the elements of the Kripke Structures.

```
module Main {
  inputs: enable: bool;
  statevars: pc : {A, B, C};
  sharedvars: timerCount : integer;
  init:
      timerCount = 2000 && pc = A
  trans:
    enable =>
        pc = A =>
             timerCount = 0  => next(pc) = C
          && timerCount != 0 => next(pc) = B
     && pc = B =>
          next(pc) = A
     && pc = C =>
          next(pc) = C
  && !enable =>
          next(pc) = pc
}

module ISR {
  inputs: enable : bool;
  outputs: return : bool;
  statevars: pc : {idle, D, E};
  sharedvars: timerCount : integer;
  init:
      pc = idle
  trans:
     enable =>
        pc = idle =>
           next(pc) = D && !return
     && pc = D && timerCount != 0 =>
           next(pc) = E && !return
     && pc = D && timerCount = 0 =>
           next(pc) = idle && return
     && pc = E =>
           next(timerCount) = timerCount-1
           && next(pc) = idle && return
  && !enable =>
        next(pc) = pc && !return
}
```

```
module System {
  inputs: assert : bool;
  statevars: mode : {main, ISR};
  wires: M_enable, I_enable: bool;
  composition: synchronous;
  instances:
     M : main; I : ISR;
  connect:
     M.timerCount = I.timerCount;
     M.enable = M_enable;
     I.enable = I_enable;
  init:
      mode = main
  trans:
     M_enable = (mode = main)
  && I_enable = (mode = ISR)
  && (mode = main || next(mode) = main)
     && assert =>
         next(mode) = ISR
  && mode = ISR && I.return =>
       next(mode) = main
}


module Environment {
  outputs: assert : bool;
  init: true
  trans: true
}


module System_and_Env {
  composition: asynchronous;
  instances:
     Sys : System,
     Env : Environment;
  connect:
     Sys.assert = Env.assert
}
```

Fig. 13 A simple interrupt-driven program modeled in SML

3.5.1 Transition Systems

Transition systems are the most common formalism used in formal verification since they naturally capture the dynamics of a discrete system. Transition systems are directed graphs where nodes model *states*, and the edges represent *transitions* denoting the state changes. A state encapsulates information of the system (i.e., values of the system variables) at a particular moment in time during its execution. For instance, a state of the mutual exclusion protocol can indicate the critical or noncritical sections of the system processes. Similarily, for example in hardware circuits executing synchronously, a state can represent the register values in addition to the values of the input bits. Transitions encapsulate the gradual changes that the system parameters exhibit at each execution step of the system. In the case of the mutual exclusion protocol a transition may indicate that a process moves from its non-critical section to a waiting or critical section state. In software, on the other hand, the transition may correspond to the execution of a program statement (say, an assignment operation) which may result in a change of the values of some program variables together with a program counter. Correspondingly in hardware a transition models the update of the registers and the output bits in response to the updated set of inputs.

There exist many classes of transition systems. The choice of a particular transition system depends on the nature of the system being modeled. This chapter presents the most common formalism, called Kripke Structures, which is suitable for modeling most hardware and software systems. Kripke structures are transition systems which are specified by constructs called *atomic propositions*. Atomic propositions express facts about the states of the system, for example, "**Heli.coeff = 10.0**" for variable **Heli.coeff** from the helicopter control system example.

Definition 2 (Kripke Structure) (cf. [25]) Let AP be a non-empty set of atomic propositions. A Kripke structure is a transition system defined as a four-tuple $M = (S; S_0; R; L)$, where S is a finite set of states, $S_0 \in S$ is a finite set of initial states ($S_0 \subseteq S$), $R \subseteq S \times S$ is a transition relation, for which it holds that $\forall s \in S : \exists s' \in S : (s; s') \in R$, and $L : S \to 2^{AP}$ is the labeling function which labels each state with the atomic propositions which hold in that state.

A path of the Kripke structure is a sequence of states $s_0, s_1, s_2, \ldots$ such that $s_0 \in S_0$ and for each $i \geq 0$, $s_{i+1} = R(s_i)$.

The word on the path is a sequence of sets of the atomic propositions $\omega = L(s_1), L(s_2), L(s_3), \ldots$, which is an ω-word over alphabet 2^{AP}.

The program semantics is defined by its language which is the set of finite (infinite) words of all possible paths which the system can take during its execution.

Kripke structures are the models defining the semantics (definition of when a specified property holds) of the most widely used specification languages for reactive systems, namely temporal logics.

Kripke structures can be seen as describing the behavior of the modeled system in a modeling-language-independent manner. Therefore, temporal logics are really modeling-formalism independent. The definition of atomic propositions is the only thing that needs to be adjusted for each formalism.

A fair execution of the program modeled by a Kripke structure is ensured by fairness constraints which rule out the unrealistic paths. A fair path ensures that certain fairness constraints hold. In general, a strong fairness constraint can be defined as a temporal logic formula of the following form: $\mathbf{GF}\ AP \Longrightarrow \mathbf{F}\ AP$, where AP is a set of atomic propositions of interest. The formula states that if some atomic propositions are true periodically (corresponding to the fact that a process is ready to execute or transition to a new state) then they will be true infinitely often in the future as well (the transitions will be taken). The weak fairness constraint is similarily defined as $\mathbf{G}\ AP \Longrightarrow \mathbf{F}\ AP$, stating that every process that is continuously ready to execute from some time point gets its turn infinitely often.

3.5.2 From SML *Programs to Kripke Structures*

For purposes of verification, we must close the model of the system under verification with the model of its environment. For such a closed SML program, suppose that

the corresponding STS is $(\emptyset, O, V, \alpha, \delta)$. From this STS, we obtain the corresponding Kripke structure as (S, S_0, R, L), where $S = 2^{V \cup O}$, $S_0 = \{s \in S \mid \alpha(s) = \textbf{true}\}$, $R = \delta$, and L is such that $L(s)$ is the value of variables in $V \cup O$ in state s.

3.6 Summary

This chapter has reviewed some of the fundamental issues in system modeling for verification. We introduced SML, a simple modeling language for abstract state machines or transition systems, and illustrated its use on three examples from different domains. We also gave semantics to SML using the well-known formalism of Kripke structures. Other chapters in this Handbook will cover several concepts in model checking, many of which are based on domain-specific modeling formalisms. SML captures the essential features of those formalisms and potentially provides a basis for better understanding the connections between chapters and even creating new connections.

References

1. Alpern, B., Schneider, F.B.: Recognizing safety and liveness. Distrib. Comput. **2**(3), 117–126 (1987)
2. Alur, R., Courcoubetis, C., Dill, D.: Model checking in dense real time. Inf. Comput. **104**(1), 2–34 (1993)
3. Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T., Ho, P., Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: The algorithmic analysis of hybrid systems. Theor. Comput. Sci. **138**, 3–34 (1995)
4. Alur, R., Dill, D.: A theory of timed automata. Theor. Comput. Sci. **126**, 183–235 (1994)
5. Alur, R., Henzinger, T.: Logics and models of real time: a survey. In: Real Time: Theory in Practice. LNCS, vol. 600 (1992)
6. Alur, R., Henzinger, T.: Reactive modules. Form. Methods Syst. Des. **15**, 7–48 (1999)
7. Baier, C., Haverkort, B., Hermanns, H., Katoen, J.P.: Performance evaluation and model checking join forces. Commun. ACM **53**(9), 76–85 (2010)
8. Baier, C., Haverkort, B., Hermanns, H., Katoen, J.P., Siegle, M. (eds.): Validation of Stochastic Systems—A Guide to Current Research. LNCS, vol. 2925. Springer, Heidelberg (2004)
9. Baier, C., Majster-Cederbaum, M.: Denotational semantics in the CPO and metric approach. Theor. Comput. Sci. **135**(2), 171–220 (1994)
10. Balarin, F., Watanabe, Y., Hsieh, H., Lavagno, L., Passerone, R., Sangiovanni-Vincentelli, A.: Metropolis: an integrated electronic system design environment. IEEE Comput. **36**, 45–52 (2003)
11. Barrett, C., Sebastiani, R., Seshia, S., Tinelli, C.: Satisfiability modulo theories. In: Biere, A., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, vol. 4. IOS Press, Amsterdam (2009). Chap. 8
12. Behrmann, G., Larsen, K., Rasmussen, J.: Priced timed automata: algorithms and applications. In: Third International Symposium on Formal Methods for Components and Objects (FMCO), pp. 162–182 (2004)
13. Benveniste, A., Caspi, P., Lublinerman, R., Tripakis, S.: Actors without directors: a Kahnian view of heterogeneous systems. In: HSCC'09: Proceedings of the 12th International Conference on Hybrid Systems: Computation and Control. LNCS, pp. 46–60. Springer, Heidelberg (2009). doi:10.1007/978-3-642-00602-9_4

14. Berry, G., Gonthier, G.: The Esterel synchronous programming language: design, semantics, implementation. Sci. Comput. Program. **19**(2), 87–152 (1992)
15. Brady, B., Bryant, R., Seshia, S.: Abstracting RTL designs to the term level. Tech. Rep. UCB/EECS-2008-136, EECS Department, University of California, Berkeley (2008) http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-136.html
16. Brady, B., Bryant, R., Seshia, S., O'Leary, J.: ATLAS: automatic term-level abstraction of RTL designs. In: Proceedings of the Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE) (2010)
17. Broman, D., Lee, E., Tripakis, S., Törngren, M.: Viewpoints, formalisms, languages, and tools for cyber-physical systems. In: 6th International Workshop on Multi-paradigm Modeling (MPM'12) (2012)
18. Broy, M., Stolen, K.: Specification and Development of Interactive Systems. Monographs in Computer Science, vol. 62. Springer, Heidelberg (2001)
19. Bryant, R.: Graph-based algorithms for Boolean function manipulation. IEEE Trans. Comput. **C-35**(8), 677–691 (1986)
20. Bryant, R., Lahiri, S., Seshia, S.: Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In: Brinksma, E., Larsen, K. (eds.) Proc. Computer-Aided Verification (CAV'02). LNCS, vol. 2404, pp. 78–92 (2002)
21. Buck, J.: Scheduling dynamic dataflow graphs with bounded memory using the token flow model. Ph.D. thesis, University of California, Berkeley (1993)
22. Caspi, P., Pilaud, D., Halbwachs, N., Plaice, J.: Lustre: a declarative language for programming synchronous systems. In: 14th ACM Symp. POPL. ACM, New York (1987)
23. Chatterjee, K., Doyen, L., Henzinger, T.: Quantitative languages. In: Proc. Computer Science Logic (CSL). LNCS, vol. 5213, pp. 385–400 (2008)
24. Chatterjee, K., Doyen, L., Henzinger, T.: Alternating weighted automata. In: Fundamentals of Computation Theory (FCT). LNCS, vol. 5699, pp. 3–13 (2009)
25. Clarke, E., Grumberg, O., Peled, D. (eds.): Model Checking. MIT Press, Cambridge (2001)
26. Commoner, F., Holt, A.W., Even, S., Pnueli, A.: Marked directed graphs. J. Comput. Syst. Sci. **5**, 511–523 (1971)
27. Cruz, R.L.: A calculus for network delay, part I. Network elements in isolation. IEEE Trans. Inf. Theory **37**(1), 114–131 (1991)
28. Damm, W., Harel, D.: LSCs: breathing life into message sequence charts. Form. Methods Syst. Des. **19**(1), 45–80 (2001)
29. Davare, A., Densmore, D., Meyerowitz, T., Pinto, A., Sangiovanni-Vincentelli, A., Yang, G., Zeng, H., Zhu, Q.: A next-generation design framework for platform-based design. In: Conference on Using Hardware Design and Verification Languages (DVCon), vol. 152 (2007)
30. Davis, M.: Markov Models and Optimization. Chapman & Hall, London (1993)
31. Daws, C., Olivero, A., Tripakis, S., Yovine, S.: The tool KRONOS. In: Alur, R., Henzinger, T., Sontag, E. (eds.) Hybrid Systems III: Verification and Control. LNCS, vol. 1066, pp. 208–219. Springer, Heidelberg (1996)
32. Eker, J., Janneck, J., Lee, E., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., Xiong, Y.: Taming heterogeneity—the Ptolemy approach. Proc. IEEE **91**(1), 127–144 (2003)
33. Fokkink, W.: Introduction to Process Algebra. Springer, Heidelberg (2000)
34. Gurevich, Y., Kutter, P.W., Odersky, M., Thiele, L. (eds.): Abstract State Machines, Theory and Applications, Proceedings of the International Workshop, ASM 2000, Monte Verità, Switzerland, March 19–24, 2000. LNCS, vol. 1912. Springer, Heidelberg (2000)
35. Harel, D.: Statecharts: a visual formalism for complex systems. Sci. Comput. Program. **8**, 231–274 (1987)
36. Hoare, C.: Communicating Sequential Processes. Prentice Hall, New York (1985)
37. Holcomb, D., Brady, B., Seshia, S.: Abstraction-based performance analysis of NoCs. In: Proceedings of the Design Automation Conference (DAC), pp. 492–497 (2011)
38. Hopcroft, J., Motwani, R., Ullman, J.: Introduction to Automata Theory, Languages, and Computation, 3rd edn. Addison-Wesley, Reading (2006)

39. Hu, J., Lygeros, J., Sastry, S.: Towards a theory of stochastic hybrid systems. In: Hybrid Systems: Computation and Control (HSCC). LNCS, vol. 1790, pp. 160–173. Springer, Heidelberg (2000)
40. ITU: Z.120—Message Sequence Chart (MSC). Available at http://www.itu.int/rec/T-REC-Z.120 (02/2011)
41. ITU: Z.120 Annex B: Formal semantics of Message Sequence Charts. Available at http://www.itu.int/rec/T-REC-Z.120 (04/1998)
42. Kahn, G.: The semantics of a simple language for parallel programming. In: Information Processing 74. Proceedings of IFIP Congress, vol. 74. North-Holland, Amsterdam (1974)
43. Karp, R., Miller, R.: Properties of a model for parallel computations: determinacy, termination, queueing. SIAM J. Appl. Math. **14**(6), 1390–1411 (1966)
44. Kohavi, Z.: Switching and Finite Automata Theory, 2nd edn. McGraw-Hill, New York (1978)
45. Kwiatkowska, M., Norman, G., Parker, D.: Stochastic model checking. In: Bernardo, M., Hillston, J. (eds.) Formal Methods for the Design of Computer, Communication and Software Systems: Performance Evaluation (SFM'07). LNCS, vol. 4486, pp. 220–270. Springer, Heidelberg (2007)
46. Lamport, L.: Proving the correctness of multiprocess programs. IEEE Trans. Softw. Eng. **3**(2), 125–143 (1977)
47. Larsen, K., Petterson, P., Yi, W.: Uppaal in a nutshell. Software Tools for Technology Transfer **1**(1/2) (1997)
48. Lee, E., Messerschmitt, D.: Synchronous data flow. Proc. IEEE **75**(9), 1235–1245 (1987)
49. Lee, E., Seshia, S.: Introduction to Embedded Systems—A Cyber-physical Systems Approach (2011)
50. Liu, X., Lee, E.: CPO semantics of timed interactive actor networks. Theor. Comput. Sci. **409**(1), 110–125 (2008)
51. Malik, S., Zhang, L.: Boolean satisfiability: from theoretical hardness to practical success. Commun. ACM **52**(8), 76–82 (2009)
52. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems: Specification. Springer, New York (1991)
53. Milner, R.: A Calculus of Communicating Systems. LNCS, vol. 92. Springer, Heidelberg (1980)
54. Milner, R.: Communicating and Mobile Systems: The π-Calculus. Cambridge University Press, Cambridge (1999)
55. Peh, L.S.: Flow control and micro-architectural mechanisms for extending the performance of interconnection networks. Ph.D. thesis, Stanford University (2001)
56. Reisig, W.: Petri Nets: An Introduction. Springer, Heidelberg (1985)
57. Seshia, S.: Quantitative analysis of software: challenges and recent advances. In: 7th International Workshop on Formal Aspects of Component Software (FACS) (2010)
58. Stergiou, C., Tripakis, S., Matsikoudis, E., Lee, E.: On the verification of timed discrete-event models. In: FORMATS 2013. Springer, Heidelberg (2013)
59. Theelen, B., Geilen, M., Stuijk, S., Gheorghita, S., Basten, T., Voeten, J., Ghamarian, A.: Scenario-aware dataflow. Tech. Rep. ESR-2008-08, Eindhoven University of Technology, (2008)
60. Tripakis, S.: Compositionality in the science of system design. Proc. IEEE **104**(5), 960–972 (2016)
61. Tripakis, S., Stergiou, C., Shaver, C., Lee, E.: A modular formal semantics for Ptolemy. Math. Struct. Comput. Sci. **23**, 834–881 (2013). doi:10.1017/S0960129512000278
62. Yates, R.: Networks of real-time processes. In: Best, E. (ed.) Proc. of the 4th Int. Conf. on Concurrency Theory (CONCUR). LNCS, vol. 715. Springer, Heidelberg (1993)

Chapter 4
Automata Theory and Model Checking

Orna Kupferman

Abstract We study automata on infinite words and their applications in system specification and verification. We first introduce Büchi automata and survey their closure properties, expressive power, and determinization. We then introduce additional acceptance conditions and the model of alternating automata. We compare the different classes of automata in terms of expressive power and succinctness, and describe decision problems for them. Finally, we describe the automata-theoretic approach to system specification and verification.

4.1 Introduction

Finite automata on infinite objects were first introduced in the 1960s. Motivated by decision problems in mathematics and logic, Büchi, McNaughton, and Rabin developed a framework for reasoning about infinite words and infinite trees [6, 52, 61]. The framework has proved to be very powerful. Automata and their tight relation to second-order monadic logics were the key to the solution of several fundamental decision problems in mathematics and logic [62, 74]. Today, automata on infinite objects are used for specification and verification of nonterminating systems. The idea is simple: when a system is defined with respect to a finite set AP of propositions, each of the system's states can be associated with a set of propositions that hold in this state. Then, each of the system's computations induces an infinite word over the alphabet 2^{AP}, and the system itself induces a language of infinite words over this alphabet. This language can be defined by an automaton. Similarly, a system specification, which describes all the allowed computations, can be viewed as a language of infinite words over 2^{AP}, and can therefore be defined by an automaton. In the automata-theoretic approach to verification, we reduce questions about systems and their specifications to questions about automata. More specifically, questions such as satisfiability of specifications and correctness of systems with respect to their specifications are reduced to questions such as non-emptiness and language containment [48, 77, 79].

O. Kupferman (✉)
Hebrew University, Jerusalem, Israel
e-mail: orna@cs.huji.ac.il

E.M. Clarke et al. (eds.), *Handbook of Model Checking*,
DOI 10.1007/978-3-319-10575-8_4

The automata-theoretic approach separates the logical and the combinatorial aspects of reasoning about systems. The translation of specifications to automata handles the logic and shifts all the combinatorial difficulties to automata-theoretic problems, yielding clean and asymptotically optimal algorithms, as well as better understanding of the complexity of the problems. Beyond leading to tight complexity bounds, automata have proven to be very helpful in practice. Automata are the key to techniques such as on-the-fly model checking [11, 21], and they are useful also for modular model checking [41], partial-order model checking [23, 31, 75, 78], model checking of real-time and hybrid systems [26], open systems [1], and infinite-state systems [40, 43]. Automata also serve as expressive specification formalisms [2, 39] and in algorithms for sanity checks [37]. Automata-based methods have been implemented in both academic and industrial automated-verification tools (e.g., COSPAN [24], SPIN [27], ForSpec [72], and NuSMV [9]).

This chapter studies automata on infinite words and their applications in system specification and verification. We first introduce Büchi automata, survey their closure properties, expressive power, and determinization. We then introduce additional acceptance conditions and the model of alternating automata. We compare the different classes of automata in terms of expressive power and succinctness, and describe decision problems for them. Finally, we describe the automata-theoretic approach to system specification and verification.

4.2 Nondeterministic Büchi Automata on Infinite Words

4.2.1 Definitions

For a finite alphabet Σ, an infinite word $w = \sigma_1 \cdot \sigma_2 \cdot \sigma_3 \cdots$ is an infinite sequence of letters from Σ. We use Σ^ω to denote the set of all infinite words over the alphabet Σ. A language $L \subseteq \Sigma^\omega$ is a set of words. We sometimes refer also to finite words, and to languages $L \subseteq \Sigma^*$ of finite words over Σ. A *prefix* of $w = \sigma_1 \cdot \sigma_2 \cdots$ is a (possibly empty) finite word $\sigma_1 \cdot \sigma_2 \cdot \sigma_3 \cdots \sigma_i$, for some $i \geq 0$. A *suffix* of w is an infinite word $\sigma_i \cdot \sigma_{i+1} \cdots$, for some $i \geq 1$. A property of a system with a set AP of atomic propositions can be viewed as a language over the alphabet 2^{AP}. We have seen in Chap. 2 that languages over this alphabet can be defined by linear temporal-logic (LTL, for short) formulas. Another way to define languages is by automata.

A *nondeterministic finite automaton* is a tuple $\mathcal{A} = \langle \Sigma, Q, Q_0, \delta, \alpha \rangle$, where Σ is a finite non-empty alphabet, Q is a finite non-empty set of *states*, $Q_0 \subseteq Q$ is a non-empty set of *initial states*, $\delta : Q \times \Sigma \to 2^Q$ is a *transition function*, and α is an *acceptance condition*, to be defined below.

Intuitively, when the automaton $\mathcal{A}$ runs on an input word over Σ, it starts in one of the initial states, and it proceeds along the word according to the transition function. Thus, $\delta(q, \sigma)$ is the set of states that $\mathcal{A}$ can move into when it is in state q and it reads the letter σ. Note that the automaton may be *nondeterministic*, since it may have several initial states and the transition function may specify several

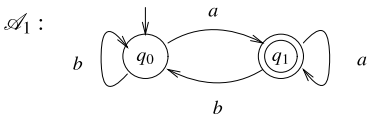


Fig. 1 A DBW for $\{w : w \text{ has infinitely many } a\text{'s}\}$

possible transitions for each state and letter. The automaton $\mathscr{A}$ is *deterministic* if $|Q_0| = 1$ and $|\delta(q, \sigma)| = 1$ for all states $q \in Q$ and symbols $\sigma \in \Sigma$. Specifying deterministic automata, we sometimes describe the single initial state or destination state, rather than a singleton set.

Formally, a *run* r of $\mathscr{A}$ on a finite word $w = \sigma_1 \cdot \sigma_2 \cdots \sigma_n \in \Sigma^*$ is a sequence $r = q_0, q_1, \ldots, q_n$ of $n + 1$ states in Q such that $q_0 \in Q_0$, and for all $0 \leq i < n$ we have $q_{i+1} \in \delta(q_i, \sigma_{i+1})$. Note that a nondeterministic automaton may have several runs on a given input word. In contrast, a deterministic automaton has exactly one run on a given input word. When the input word is infinite, thus $w = \sigma_1 \cdot \sigma_2 \cdot \sigma_3 \cdots \in \Sigma^\omega$, then a run of $\mathscr{A}$ on it is an infinite sequence of states $r = q_0, q_1, q_2, \ldots$ such that $q_0 \in Q_0$, and for all $i \geq 0$, we have $q_{i+1} \in \delta(q_i, \sigma_{i+1})$. For an infinite run r, let $inf(r) = \{q : q_i = q \text{ for infinitely many } i\text{'s }\}$. Thus, $inf(r)$ is the set of states that r visits infinitely often.

The acceptance condition α determines which runs are "good". For automata on finite words, $\alpha \subseteq Q$ and a run r is *accepting* if $q_n \in \alpha$. For automata on infinite words, one can consider several acceptance conditions. Let us start with the Büchi acceptance condition [6]. There, $\alpha \subseteq Q$, and a run r is accepting if it visits some state in α infinitely often. Formally, r is accepting iff $inf(r) \cap \alpha \neq \emptyset$. A run that is not accepting is *rejecting*. A word w is accepted by an automaton $\mathscr{A}$ if there is an accepting run of $\mathscr{A}$ on w. The language recognized by $\mathscr{A}$, denoted $\mathscr{L}(\mathscr{A})$, is the set of words that $\mathscr{A}$ accepts. We sometimes refer to $\mathscr{L}(\mathscr{A})$ also as the language of $\mathscr{A}$.

We use NBW and DBW to abbreviate nondeterministic and deterministic Büchi automata, respectively.[1] For a class γ of automata (so far, we have introduced $\gamma \in \{\text{NBW}, \text{DBW}\}$), we say that a language $L \subseteq \Sigma^\omega$ is γ-recognizable iff there is an automaton in the class γ that recognizes L. A language is *ω-regular* iff it is NBW-recognizable.

Example 1 Consider the DBW $\mathscr{A}_1$ appearing in Fig. 1. When we draw automata, states are denoted by circles. Directed edges between states are labeled with letters and describe the transitions. Initial states (q_0, in the figure) have an edge entering them with no source, and accepting states (q_1, in the figure) are identified by double circles. The DBW moves to the accepting state whenever it reads the letter a, and it moves to the non-accepting state whenever it reads the letter b. Accordingly, the single run r on a word w visits the accepting state infinitely often iff w has infinitely many a's. Hence, $\mathscr{L}(\mathscr{A}_1) = \{w : w \text{ has infinitely many } a\text{'s}\}$.

[1] The letter W indicates that the automata run on words (rather than, say, trees).

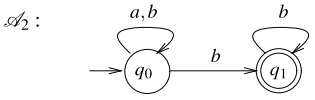


Fig. 2 An NBW for $\{w : w \text{ has only finitely many } a\text{'s}\}$

Example 2 Consider the NBW $\mathscr{A}_2$ appearing in Fig. 2. The automaton is nondeterministic, and in order for a run to be accepting it has to eventually move to the accepting state, where it has to stay forever while reading b. Note that if $\mathscr{A}_2$ reads a from the accepting state it gets stuck. Accordingly, $\mathscr{A}_2$ has an accepting run on a word w iff w has a position from which an infinite tail of b's starts. Hence, $\mathscr{L}(\mathscr{A}_2) = \{w : w \text{ has only finitely many } a\text{'s}\}$.

Consider a directed graph $G = \langle V, E \rangle$. A *strongly connected set* of G (SCS) is a set $C \subseteq V$ of vertices such that for every two vertices $v, v' \in C$, there is a path from v to v'. An SCS C is *maximal* if it cannot be extended to a larger SCS. Formally, for every nonempty $C' \subseteq V \setminus C$, we have that $C \cup C'$ is not an SCS. The maximal strongly connected sets are also termed *strongly connected components* (SCC). An automaton $\mathscr{A} = \langle \Sigma, Q, Q_0, \delta, \alpha \rangle$ induces a directed graph $G_{\mathscr{A}} = \langle Q, E \rangle$ in which $\langle q, q' \rangle \in E$ iff there is a letter σ such that $q' \in \delta(q, \sigma)$. When we talk about the SCSs and SCCs of $\mathscr{A}$, we refer to those of $G_{\mathscr{A}}$. Consider a run r of an automaton $\mathscr{A}$. It is not hard to see that the set $inf(r)$ is an SCS. Indeed, since every two states q and q' in $inf(r)$ are visited infinitely often, the state q' must be reachable from q.

4.2.2 Closure Properties

Automata on finite words are closed under union, intersection, and complementation. In this section we study closure properties for nondeterministic Büchi automata.

4.2.2.1 Closure Under Union and Intersection

We start with closure under union, where the construction that works for nondeterministic automata on finite words, namely putting the two automata "one next to the other", works also for nondeterministic Büchi automata. Formally, we have the following.

Theorem 1 ([8]) *Let $\mathscr{A}_1$ and $\mathscr{A}_2$ be NBWs with n_1 and n_2 states, respectively. There is an NBW $\mathscr{A}$ such that $\mathscr{L}(\mathscr{A}) = \mathscr{L}(\mathscr{A}_1) \cup \mathscr{L}(\mathscr{A}_2)$ and $\mathscr{A}$ has $n_1 + n_2$ states.*

Proof Let $\mathscr{A}_1 = \langle \Sigma, Q_1, Q_1^0, \delta_1, \alpha_1 \rangle$ and $\mathscr{A}_2 = \langle \Sigma, Q_2, Q_2^0, \delta_2, \alpha_2 \rangle$. We assume, without loss of generality, that Q_1 and Q_2 are disjoint. Since nondeterministic automata may have several initial states, we can define $\mathscr{A}$ as the NBW obtained by

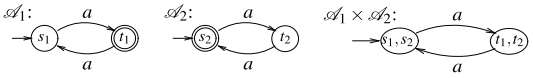


Fig. 3 Two Büchi automata accepting the language $\{a^\omega\}$, and their empty product

taking the union of $\mathscr{A}_1$ and $\mathscr{A}_2$. Thus, $\mathscr{A} = \langle \Sigma, Q_1 \cup Q_2, Q_1^0 \cup Q_2^0, \delta, \alpha_1 \cup \alpha_2 \rangle$, where for every state $q \in Q_1 \cup Q_2$, we have that $\delta(q, \sigma) = \delta_i(q, \sigma)$, for the index $i \in \{1, 2\}$ such that $q \in Q_i$. It is easy to see that for every word $w \in \Sigma^\omega$, the NBW $\mathscr{A}$ has an accepting run on w iff at least one of the NBWs $\mathscr{A}_1$ and $\mathscr{A}_2$ has an accepting run on w. □

We proceed to closure under intersection. For the case of finite words, one proves closure under intersection by constructing, given $\mathscr{A}_1$ and $\mathscr{A}_2$, a "product automaton" that has $Q_1 \times Q_2$ as its state space and simulates the runs of both $\mathscr{A}_1$ and $\mathscr{A}_2$ on the input words. A word is then accepted by both $\mathscr{A}_1$ and $\mathscr{A}_2$ iff the product automaton has a run that leads to a state in $\alpha_1 \times \alpha_2$. As the example below demonstrates, this construction does not work for Büchi automata.

Example 3 Consider the two DBWs $\mathscr{A}_1$ and $\mathscr{A}_2$ on the left of Fig. 3. The product automaton $\mathscr{A}_1 \times \mathscr{A}_2$ is shown on the right. Clearly, $\mathscr{L}(\mathscr{A}_1) = \mathscr{L}(\mathscr{A}_2) = \{a^\omega\}$, but $\mathscr{L}(\mathscr{A}_1 \times \mathscr{A}_2) = \emptyset$.

As demonstrated in Example 3, the problem with the product automaton is that the definition of the set of accepting states to be $\alpha_1 \times \alpha_2$ forces the accepting runs of $\mathscr{A}_1$ and $\mathscr{A}_2$ to visit α_1 and α_2 simultaneously. This requirement is too strong, as an input word may still be accepted by both $\mathscr{A}_1$ and $\mathscr{A}_2$, but the accepting runs on it visit α_1 and α_2 in different positions. As we show below, the product automaton is a good basis for proving closure under intersection, but one needs to take two copies of it: one that waits for visits of runs of $\mathscr{A}_1$ to α_1 (and moves to the second copy when such a visit is detected) and one that waits for visits of runs of $\mathscr{A}_2$ to α_2 (and returns to the first copy when such a visit is detected). The acceptance condition then requires the run to alternate between the two copies infinitely often, which is possible exactly when both the run of $\mathscr{A}_1$ visits α_1 infinitely often, and the run of $\mathscr{A}_2$ visits α_2 infinitely often. Note that $\mathscr{A}_2$ may visit α_2 when the run is in the first copy, in which case the visit to α_2 is ignored, and in fact this may happen infinitely many times. Still, if there are infinitely many visits to α_1 and α_2, then eventually the run moves to the second copy, where it eventually comes across a visit to α_2 that is not ignored. Formally, we have the following.

Theorem 2 ([8]) *Let $\mathscr{A}_1$ and $\mathscr{A}_2$ be NBWs with n_1 and n_2 states, respectively. There is an NBW $\mathscr{A}$ such that $\mathscr{L}(\mathscr{A}) = \mathscr{L}(\mathscr{A}_1) \cap \mathscr{L}(\mathscr{A}_2)$ and $\mathscr{A}$ has $2n_1n_2$ states.*

Proof Let $\mathscr{A}_1 = \langle \Sigma, Q_1, Q_1^0, \delta_1, \alpha_1 \rangle$ and $\mathscr{A}_2 = \langle \Sigma, Q_2, Q_2^0, \delta_2, \alpha_2 \rangle$. We define $\mathscr{A} = \langle \Sigma, Q, Q^0, \delta, \alpha \rangle$, where

- $Q = Q_1 \times Q_2 \times \{1, 2\}$. That is, the state space consists of two copies of the product automaton.

- $Q^0 = Q_1^0 \times Q_2^0 \times \{1\}$. That is, the initial states are triples $\langle s_1, s_2, 1\rangle$ such that s_1 and s_2 are initial in $\mathcal{A}_1$ and $\mathcal{A}_2$, respectively. The run starts in the first copy.
- For all $q_1 \in Q_1$, $q_2 \in Q_2$, $c \in \{1, 2\}$, and $\sigma \in \Sigma$, we define $\delta(\langle s_1, s_2, c\rangle, \sigma) = \delta_1(s_1, \sigma) \times \delta_2(s_2, \sigma) \times \{next(s_1, s_2, c)\}$, where

$$next(s_1, s_2, c) = \begin{cases} 1 & \text{if } (c = 1 \text{ and } s_1 \notin \alpha_1) \text{ or } (c = 2 \text{ and } s_2 \in \alpha_2), \\ 2 & \text{if } (c = 1 \text{ and } s_1 \in \alpha_1) \text{ or } (c = 2 \text{ and } s_2 \notin \alpha_2). \end{cases}$$

  That is, $\mathcal{A}$ proceeds according to the product automaton, and it moves from the first copy to the second copy when $s_1 \in \alpha_1$, and from the second copy to the first copy when $s_2 \in \alpha_2$. In all other cases it stays in the current copy.
- $\alpha = \alpha_1 \times Q_2 \times \{1\}$. That is, a run of $\mathcal{A}$ is accepting if it visits infinitely many states in the first copy in which the Q_1-component is in α_1. Note that after such a visit, $\mathcal{A}$ moves to the second copy, from which it returns to the first copy after visiting a state in which the Q_2-component is in α_2. Accordingly, there must be a visit to a state in which the Q_2-component is in α_2 between every two successive visits to states in α. This is why a run visits α infinitely often iff its Q_1-component visits α_1 infinitely often and its Q_2-component visits α_2 infinitely often. □

Note that the product construction retains determinism; i.e., starting with deterministic $\mathcal{A}_1$ and $\mathcal{A}_2$, the product $\mathcal{A}$ is deterministic. Thus, DBWs are also closed under intersection. Also, while the union construction we have described does not retain determinism, DBWs are closed also under union. Indeed, if we take the product construction (one copy of it is sufficient), which retains determinism, and define the set of accepting states to be $(\alpha_1 \times Q_2) \cup (Q_1 \times \alpha_2)$, we get a DBW for the union. Note, however, that unlike the $n_1 + n_2$ blow-up in Theorem 1, the blow-up now is $n_1 n_2$.

4.2.2.2 Closure Under Complementation

For deterministic automata on finite words, complementation is easy: the single run is rejecting iff its last state is not accepting, thus complementing a deterministic automaton can proceed by *dualizing* its acceptance condition: for an automaton with state space Q and set α of accepting states, the dual acceptance condition is $\tilde{\alpha} = Q \setminus \alpha$, and it is easy to see that dualizing the acceptance condition of a deterministic automaton on finite words results in a deterministic automaton for the complement language. It is also easy to see that such a simple dualization does not work for DBWs. Indeed, a run of a Büchi automaton is rejecting iff it visits α only finitely often, which is different from requiring it to visit $\tilde{\alpha}$ infinitely often. As a concrete example, consider the DBW $\mathcal{A}_1$ from Fig. 1. Recall that $\mathcal{L}(\mathcal{A}_1) = \{w : w$ has infinitely many a's$\}$. An attempt to complement it by defining the set of accepting states to be $\{q_0\}$ results in a DBW whose language is $\{w : w$ has infinitely many b's$\}$, which does not complement $\mathcal{L}(\mathcal{A}_1)$. For example, the word $(a \cdot b)^\omega$ belongs to both languages. In this section we study the complementation problem for

Büchi automata. We start with deterministic automata and show that while dualization does not work, their complementation is quite simple, but results in a nondeterministic automaton. We then move on to nondeterministic automata, and describe a complementation procedure for them.

Theorem 3 ([47]) *Let $\mathscr{A}$ be a DBW with n states. There is an NBW $\mathscr{A}'$ such that $\mathscr{L}(\mathscr{A}') = \Sigma^\omega \setminus \mathscr{L}(\mathscr{A})$, and $\mathscr{A}'$ has at most $2n$ states.*

Proof Let $\mathscr{A} = \langle \Sigma, Q, q_0, \delta, \alpha \rangle$. The NBW $\mathscr{A}'$ should accept exactly all words w for which the single run of $\mathscr{A}$ on w visits α only finitely often. It does so by guessing a position from which no more visits of $\mathscr{A}$ to α take place. For that, $\mathscr{A}'$ consists of two copies of $\mathscr{A}$: one that includes all the states and transitions of $\mathscr{A}$, and one that excludes the accepting states of $\mathscr{A}$, and to which $\mathscr{A}'$ moves when it guesses that no more states in α are going to be visited. All the states in the second copy are accepting. Formally, $\mathscr{A}' = \langle \Sigma, Q', Q'_0, \delta', \alpha' \rangle$, where

- $Q' = (Q \times \{0\}) \cup ((Q \setminus \alpha) \times \{1\})$.
- $Q'_0 = \{\langle q_0, 0\rangle\}$.
- For every $q \in Q$, $c \in \{0, 1\}$, and $\sigma \in \Sigma$ with $\delta(q, \sigma) = q'$, we have

$$\delta'(\langle q, c\rangle, \sigma) = \begin{cases} \{\langle q', 0\rangle, \langle q', 1\rangle\} & \text{if } c = 0 \text{ and } q' \notin \alpha, \\ \{\langle q', 0\rangle\} & \text{if } c = 0 \text{ and } q' \in \alpha, \\ \{\langle q', 1\rangle\} & \text{if } c = 1 \text{ and } q' \notin \alpha, \\ \emptyset & \text{if } c = 1 \text{ and } q' \in \alpha. \end{cases}$$

- $\alpha' = (Q \setminus \alpha) \times \{1\}$.

Thus, $\mathscr{A}'$ can stay in the first copy forever, but in order for a run of $\mathscr{A}'$ to be accepting, it must eventually move to the second copy, from where it cannot go back to the first copy and must avoid states in α. □

The construction described in the proof of Theorem 3 can be applied also to nondeterministic automata. Since, however, $\mathscr{A}'$ accepts a word w iff there exists a run of $\mathscr{A}$ on w that visits α only finitely often, whereas a complementing automaton should accept a word w iff all the runs of $\mathscr{A}$ on w visit α only finitely often, the construction has a one-sided error when applied to nondeterministic automata. This is not surprising, as the same difficulty exists when we complement nondeterministic automata on finite words. By restricting attention to deterministic automata, we guarantee that the existential and universal quantification on the runs of $\mathscr{A}$ coincide.

We now turn to consider complementation for nondeterministic Büchi automata. In the case of finite words, one first determinizes the automaton and then complements the result. An attempt to follow a similar plan for NBWs, namely a translation to a DBW and then an application of Theorem 3, does not work: as we shall see in Sect. 4.2.3, DBWs are strictly less expressive than NBWs, thus not all NBWs can be determinized. Nevertheless, NBWs are closed under complementation.

Efforts to develop a complementation construction for NBWs started in the early 1960s, motivated by decision problems for second-order logics. Büchi introduced

a complementation construction that involved a complicated Ramsey-based combinatorial argument and a doubly-exponential blow-up in the state space [6]. Thus, complementing an NBW with n states resulted in an NBW with $2^{2^{O(n)}}$ states. In [70], Sistla et al. suggested an improved implementation of Büchi's construction, with only $2^{O(n^2)}$ states, which is still not optimal.[2] Only in [64], Safra introduced a determinization construction that involves an acceptance condition that is stronger than Büchi, and used it in order to present a $2^{O(n \log n)}$ complementation construction, matching the known lower bound [54]. The use of complementation in practice has led to a resurgent interest in the exact blow-up that complementation involves and the feasibility of the complementation construction (e.g., issues like whether the construction can be implemented symbolically, whether it is amenable to optimizations or heuristics—these are all important criteria that complementation constructions that involve determinization do not satisfy). In [33], Klarlund introduced an optimal complementation construction that avoids determinization. Rather, the states of the complementing automaton utilize *progress measures*—a generic concept for quantifying how each step of a system contributes to bringing a computation closer to its specification. In [44], Kupferman and Vardi used ranks, which are similar to progress measures, in a complementation construction that goes via intermediate alternating co-Büchi automata. Below we describe the construction of [44] circumventing the intermediate alternating automata.

Let $\mathscr{A} = \langle \Sigma, Q, Q_0, \delta, \alpha \rangle$ be an NBW with n states. Let $w = \sigma_1 \cdot \sigma_2 \cdot \sigma_3 \cdots$ be a word in Σ^ω. We define an infinite DAG G that embodies all the possible runs of $\mathscr{A}$ on w. Formally, $G = \langle V, E \rangle$, where

- $V \subseteq Q \times \mathbb{N}$ is the union $\bigcup_{l \geq 0}(Q_l \times \{l\})$, where for all $l \geq 0$, we have $Q_{l+1} = \bigcup_{q \in Q_l} \delta(q, \sigma_{l+1})$.
- $E \subseteq \bigcup_{l \geq 0}(Q_l \times \{l\}) \times (Q_{l+1} \times \{l+1\})$ is such that for all $l \geq 0$, we have $E(\langle q, l \rangle, \langle q', l+1 \rangle)$ iff $q' \in \delta(q, \sigma_{l+1})$.

We refer to G as the *run* DAG of $\mathscr{A}$ on w. We say that a vertex $\langle q', l' \rangle$ is a *successor* of a vertex $\langle q, l \rangle$ iff $E(\langle q, l \rangle, \langle q', l' \rangle)$. We say that $\langle q', l' \rangle$ is *reachable* from $\langle q, l \rangle$ iff there exists a sequence $\langle q_0, l_0 \rangle, \langle q_1, l_1 \rangle, \langle q_2, l_2 \rangle, \ldots$ of successive vertices such that $\langle q, l \rangle = \langle q_0, l_0 \rangle$, and there exists $i \geq 0$ such that $\langle q', l' \rangle = \langle q_i, l_i \rangle$. We say that a vertex $\langle q, l \rangle$ is an *α-vertex* iff $q \in \alpha$. Finally, we say that G is an *accepting* run DAG if G has a path with infinitely many α-vertices. Otherwise, we say that G is rejecting. It is easy to see that $\mathscr{A}$ accepts w iff G is accepting.

For $k \in \mathbb{N}$, let $[k]$ denote the set $\{0, 1, \ldots, k\}$. A *ranking* for G is a function $f : V \to [2n]$ that satisfies the following two conditions:

1. For all vertices $\langle q, l \rangle \in V$, if $f(\langle q, l \rangle)$ is odd, then $q \notin \alpha$.
2. For all edges $\langle \langle q, l \rangle, \langle q', l' \rangle \rangle \in E$, we have $f(\langle q', l' \rangle) \leq f(\langle q, l \rangle)$.

[2] Interestingly, by carrying out some simple optimizations, the Ramsey-based approach in the constructions in [6] and [70] can be improved to produce complementing NBWs with the optimal $2^{O(n \log n)}$ blow-up [5].

Thus, a ranking associates with each vertex in G a rank in $[2n]$ so that the ranks along paths decrease monotonically, and α-vertices get only even ranks. Note that each path in G eventually gets trapped in some rank. We say that the ranking f is an *odd ranking* if all the paths of G eventually get trapped in an odd rank. Formally, f is odd iff for all paths $\langle q_0, 0\rangle, \langle q_1, 1\rangle, \langle q_2, 2\rangle, \ldots$ in G, there is $j \geq 0$ such that $f(\langle q_j, j\rangle)$ is odd, and for all $i \geq 1$, we have $f(\langle q_{j+i}, j+i\rangle) = f(\langle q_j, j\rangle)$.

We are going to prove that G is rejecting iff it has an odd ranking. The difficult direction is to show that if G is rejecting, then it has an odd ranking. Below we make some observations on rejecting run DAGs that help us with this direction. We say that a vertex $\langle q, l\rangle$ is *finite* in a DAG $G' \subseteq G$ iff only finitely many vertices in G' are reachable from $\langle q, l\rangle$. The vertex $\langle q, l\rangle$ is *α-free* in G' iff all the vertices in G' that are reachable from $\langle q, l\rangle$ are not α-vertices. Note that, in particular, an α-free vertex is not an α-vertex. We define an infinite sequence $G_0 \supseteq G_1 \supseteq G_2 \supseteq \ldots$ of DAGs inductively as follows.

- $G_0 = G$.
- For $i \geq 0$, we have $G_{2i+1} = G_{2i} \setminus \{\langle q, l\rangle \ : \ \langle q, l\rangle \text{ is finite in } G_{2i}\}$.
- For $i \geq 0$, we have $G_{2i+2} = G_{2i+1} \setminus \{\langle q, l\rangle \ : \ \langle q, l\rangle \text{ is } \alpha\text{-free in } G_{2i+1}\}$.

Lemma 1 *If G is rejecting, then for every $i \geq 0$, there exists l_i such that for all $l \geq l_i$, there are at most $n - i$ vertices of the form $\langle q, l\rangle$ in G_{2i}.*

Proof We prove the lemma by an induction on i. The case where $i = 0$ follows from the definition of $G_0 = G$. Indeed, in G all levels $l \geq 0$ have at most n vertices of the form $\langle q, l\rangle$. Assume that the lemma's requirement holds for i; we prove it for $i + 1$. Consider the DAG G_{2i}. We distinguish between two cases. First, if G_{2i} is finite, then G_{2i+1} is empty, G_{2i+2} is empty as well, and we are done. Otherwise, we claim that there must be some α-free vertex in G_{2i+1}. To see this, assume, by way of contradiction, that G_{2i} is infinite and no vertex in G_{2i+1} is α-free. Since G_{2i} is infinite, G_{2i+1} is also infinite. Also, each vertex in G_{2i+1} has at least one successor. Consider some vertex $\langle q_0, l_0\rangle$ in G_{2i+1}. Since, by the assumption, it is not α-free, there exists an α-vertex $\langle q'_0, l'_0\rangle$ reachable from $\langle q_0, l_0\rangle$. Let $\langle q_1, l_1\rangle$ be a successor of $\langle q'_0, l'_0\rangle$. By the assumption, $\langle q_1, l_1\rangle$ is also not α-free. Hence, there exists an α-vertex $\langle q'_1, l'_1\rangle$ reachable from $\langle q_1, l_1\rangle$. Let $\langle q_2, l_2\rangle$ be a successor of $\langle q'_1, j'_1\rangle$. By the assumption, $\langle q_2, l_2\rangle$ is also not α-free. Thus, we can continue similarly and construct an infinite sequence of vertices $\langle q_j, l_j\rangle, \langle q'_j, l'_j\rangle$ such that for all j, the vertex $\langle q'_j, l'_j\rangle$ is an α-vertex reachable from $\langle q_j, l_j\rangle$, and $\langle q_{j+1}, l_{j+1}\rangle$ is a successor of $\langle q'_j, l'_j\rangle$. Such a sequence, however, corresponds to a path in G with infinitely many α-vertices, contradicting the assumption that G is rejecting.

So, let $\langle q, l\rangle$ be an α-free vertex in G_{2i+1}. We claim that taking $l_{i+1} = \max\{l, l_i\}$ satisfies the requirement of the lemma. That is, we claim that for all $j \geq \max\{l, l_i\}$, there are at most $n - (i + 1)$ vertices of the form $\langle q, j\rangle$ in G_{2i+2}. Since $\langle q, l\rangle$ is in G_{2i+1}, it is not finite in G_{2i}. Thus, there are infinitely many vertices in G_{2i} that are reachable from $\langle q, l\rangle$. Hence, by König's Lemma, G_{2i} contains an infinite path $\langle q, l\rangle, \langle q_1, l+1\rangle, \langle q_2, l+2\rangle, \ldots$. For all $k \geq 1$, the vertex $\langle q_k, l+k\rangle$ has infinitely

many vertices reachable from it in G_{2i} and thus, it is not finite in G_{2i}. Therefore, the path $\langle q, l\rangle, \langle q_1, l+1\rangle, \langle q_2, l+2\rangle, \ldots$ exists also in G_{2i+1}. Recall that $\langle q, l\rangle$ is α-free. Hence, being reachable from $\langle q, l\rangle$, all the vertices $\langle q_k, l+k\rangle$ in the path are α-free as well. Therefore, they are not in G_{2i+2}. It follows that for all $j \geq l$, the number of vertices of the form $\langle q, j\rangle$ in G_{2i+2} is strictly smaller than their number in G_{2i}. Hence, by the induction hypothesis, we are done. □

Note that, in particular, by Lemma 1, if G is rejecting then G_{2n} is finite. Hence the following corollary.

Corollary 1 *If G is rejecting then G_{2n+1} is empty.*

We can now prove the main lemma required for complementation, which reduces the fact that all the runs of $\mathscr{A}$ on w are rejecting to the existence of an odd ranking for the run DAG of $\mathscr{A}$ on w.

Lemma 2 *An NBW $\mathscr{A}$ rejects a word w iff there is an odd ranking for the run* DAG *of $\mathscr{A}$ on w.*

Proof Let G be the run DAG of $\mathscr{A}$ on w. We first claim that if there is an odd ranking for G, then $\mathscr{A}$ rejects w. To see this, recall that in an odd ranking, every path in G eventually gets trapped in an odd rank. Hence, as α-vertices get only even ranks, it follows that all the paths of G, and thus all the possible runs of $\mathscr{A}$ on w, visit α only finitely often.

Assume now that $\mathscr{A}$ rejects w. We describe an odd ranking for G. Recall that if $\mathscr{A}$ rejects w, then G is rejecting and thus, by Corollary 1, each vertex $\langle q, l\rangle$ in G is removed from G_j, for some $0 \leq j \leq 2n$. Thus, there is $0 \leq i \leq n$ such that $\langle q, l\rangle$ is finite in G_{2i} or α-free in G_{2i+1}. Given a vertex $\langle q, l\rangle$, we define the *rank* of $\langle q, l\rangle$, denoted $f(q, l)$, as follows.

$$f(q,l) = \begin{cases} 2i & \text{if } \langle q,l\rangle \text{ is finite in } G_{2i}. \\ 2i+1 & \text{if } \langle q,l\rangle \text{ is } \alpha\text{-free in } G_{2i+1}. \end{cases}$$

We claim that f is an odd ranking for G. First, by Lemma 1, the subgraph G_{2n} is finite. Hence, the maximal rank that a vertex can get is $2n$. Also, since an α-free vertex cannot be an α-vertex and $f(\langle q, l\rangle)$ is odd only for α-free $\langle q, l\rangle$, the first condition for f being a ranking holds. We proceed to the second condition. We first argue (and a proof proceeds easily by an induction on i) that for every vertex $\langle q, l\rangle$ in G and rank $i \in [2n]$, if $\langle q, l\rangle \notin G_i$, then $f(q, l) < i$. Now, we prove that for every two vertices $\langle q, l\rangle$ and $\langle q', l'\rangle$ in G, if $\langle q', l'\rangle$ is reachable from $\langle q, l\rangle$ (in particular, if $\langle\langle q, l\rangle, \langle q', l'\rangle\rangle \in E$), then $f(q', l') \leq f(q, l)$. Assume that $f(q, l) = i$. We distinguish between two cases. If i is even, in which case $\langle q, l\rangle$ is finite in G_i, then either $\langle q', l'\rangle$ is not in G_i, in which case, by the above claim, its rank is at most $i-1$, or $\langle q', l'\rangle$ is in G_i, in which case, being reachable from $\langle q, l\rangle$, it must be finite in G_i and have rank i. If i is odd, in which case $\langle q, l\rangle$ is α-free in G_i, then either $\langle q', l'\rangle$ is not in G_i, in which case, by the above claim, its rank is at most $i-1$, or

$\langle q', l' \rangle$ is in G_i, in which case, being reachable from $\langle q, l \rangle$, it must by α-free in G_i and have rank i.

It remains to be proved that f is an odd ranking. By the above, in every infinite path in G, there exists a vertex $\langle q, l \rangle$ such that all the vertices $\langle q', l' \rangle$ in the path that are reachable from $\langle q, l \rangle$ have $f(q', l') = f(q, l)$. We need to prove that the rank of $\langle q, l \rangle$ is odd. Assume, by way of contradiction, that the rank of $\langle q, l \rangle$ is some even i. Thus, $\langle q, l \rangle$ is finite in G_i. Then, the rank of all the vertices in the path that are reachable from $\langle q, l \rangle$ is also i, so they all belong to G_i. Since the path is infinite, there are infinitely many such vertices, contradicting the fact that $\langle q, l \rangle$ is finite in G_i. □

By Lemma 2, an NBW $\mathscr{A}'$ that complements $\mathscr{A}$ can proceed on an input word w by guessing an odd ranking for the run DAG of $\mathscr{A}$ on w. We now define such an NBW $\mathscr{A}'$ formally. We first need some definitions and notations.

A *level ranking* for $\mathscr{A}$ is a function $g : Q \to [2n] \cup \{\bot\}$, such that if $g(q)$ is odd, then $q \notin \alpha$. For two level rankings g and g', we say that g' *covers* g if for all q and q' in Q, if $g(q) \geq 0$ and $q' \in \delta(q, \sigma)$, then $0 \leq g'(q') \leq g(q)$. For a level ranking g, let $even(g)$ be the set of states that g maps to an even rank. Formally, $even(g) = \{q : g(q) \text{ is even}\}$.

Theorem 4 *Let $\mathscr{A}$ be an NBW with n states. There is an NBW $\mathscr{A}'$ such that $\mathscr{L}(\mathscr{A}') = \Sigma^\omega \setminus \mathscr{L}(\mathscr{A})$, and $\mathscr{A}'$ has at most $2^{O(n \log n)}$ states.*

Proof Let $\mathscr{A} = \langle \Sigma, Q, Q_0, \delta, \alpha \rangle$. Let $\mathscr{R}$ be the set of all level rankings for $\mathscr{A}$. When $\mathscr{A}'$ runs on a word w, it guesses an odd ranking for the run DAG of $\mathscr{A}$ on w. Each state of $\mathscr{A}'$ is a pair $\langle g, P \rangle \in \mathscr{R} \times 2^Q$. The level ranking g maintains the states in the current level of the DAG (those that are not mapped to $\bot$) and the guessed rank for them. The set P is a subset of these states, used for ensuring that all paths visit odd ranks infinitely often, which, by the definition of odd rankings, implies that all paths get stuck in some odd rank.

Formally, $\mathscr{A}' = \langle \Sigma, \mathscr{R} \times 2^Q, Q'_0, \delta', \mathscr{R} \times \{\emptyset\} \rangle$, where

- $Q'_0 = \{\langle g_0, \emptyset \rangle\}$, where $g_0(q) = 2n$ for $q \in Q_0$, and $g_0(q) = \bot$ for $q \notin Q_0$. Thus, the odd ranking that $\mathscr{A}'$ guesses maps the vertices $\langle q, 0 \rangle$ of the run DAG to $2n$.
- For a state $\langle g, P \rangle \in \mathscr{R} \times 2^Q$ and a letter $\sigma \in \Sigma$, we define $\delta'(\langle g, P \rangle, \sigma)$ as follows.
  - If $P \neq \emptyset$, then $\delta'(\langle g, P \rangle, \sigma) = \{\langle g', \delta(P, \sigma) \cap even(g') \rangle : g' \text{ covers } g\}$.
  - If $P = \emptyset$, then $\delta'(\langle g, P \rangle, \sigma) = \{\langle g', even(g') \rangle : g' \text{ covers } g\}$.

  Thus, when $\mathscr{A}'$ reads the l-th letter in the input, for $l \geq 1$, it guesses the level ranking for level l in the run DAG. This level ranking should cover the level ranking of level $l - 1$. In addition, in the P component, $\mathscr{A}'$ keeps track of states whose corresponding vertices in the DAG have even ranks. Paths that traverse such vertices should eventually reach a vertex with an odd rank. When all the paths of the DAG have visited a vertex with an odd rank, the set P becomes empty (a formal proof of the latter requires the use of König's Lemma, showing that if P does

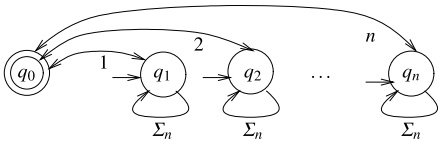


Fig. 4 The NBW $\mathscr{A}_n$

not become empty we can point to an infinite path that visits only even ranks). The set P is then initiated by new obligations for visits to vertices with odd ranks according to the current level ranking. The acceptance condition $\mathscr{R} \times \{\emptyset\}$ then checks that there are infinitely many levels in which all the obligations have been fulfilled.

Since there are $(2n+1)^n$ level rankings and 2^n subsets of Q, the automaton $\mathscr{A}'$ indeed has $2^{O(n \log n)}$ states. □

The blow-up of NBW complementation is thus $2^{O(n \log n)}$ and goes beyond the 2^n blow-up of the subset construction used in determinization and complementation of nondeterministic automata on finite words. As we see below, this blow-up cannot be avoided.

Theorem 5 ([54]) *There is a family of languages $L_1, L_2, \ldots$ such that $L_n \subseteq \Sigma_n^\omega$ can be recognized by an NBW with $n+1$ states but an NBW for $\Sigma_n^\omega \setminus L_n$ has at least $n!$ states.*

Proof For $n \geq 1$, we define L_n as the language of the NBW $\mathscr{A}_n = \langle \Sigma_n, Q_n, Q_n^0, \delta_n, \alpha \rangle$, where (see Fig. 4)

- $\Sigma_n = \{1, \ldots, n, \#\}$,
- $Q_n = \{q_0, q_1, \ldots, q_n\}$,
- $Q_n^0 = \{q_1, \ldots, q_n\}$,
- δ_n is defined as follows:

$$\delta_n(q_i, \sigma) = \begin{cases} \emptyset & \text{if } i = 0 \text{ and } \sigma = \#, \\ \{q_\sigma\} & \text{if } i = 0 \text{ and } \sigma \in \{1, \ldots, n\}, \\ \{q_i\} & \text{if } i \notin \{0, \sigma\}, \\ \{q_0, q_i\} & \text{if } \sigma = i. \end{cases}$$

- $\alpha = \{q_0\}$.

Note that a run of $\mathscr{A}_n$ is accepting if it contains infinitely many segments of the form $q_{i_1}^+ q_0 q_{i_2}^+ q_0 \cdots q_0 q_{i_k}^+ q_0 q_{i_1}^+$ for some distinct $i_1, \ldots, i_k \geq 1$. Accordingly, a word w is accepted by $\mathscr{A}_n$ iff there are k letters $\sigma_1, \sigma_2, \ldots, \sigma_k \in \{1, \ldots, n\}$, such that all the pairs $\sigma_1\sigma_2, \sigma_2\sigma_3, \ldots, \sigma_k\sigma_1$ appear in w infinitely many times. A good intuition to keep in mind is that a word $w \in \Sigma_n^\omega$ induces a directed graph $G_w = \langle \{1, \ldots, n\}, E_w \rangle$ such that $E_w(i, j)$ iff the subword $i \cdot j$ appears in w infinitely often. Then, L_n accepts exactly all words w such that G_w contains a cycle.

Consider an NBW $\mathscr{A}'_n = \langle \Sigma_n, Q'_n, {Q'}^0_n, \delta'_n, \alpha'_n \rangle$ that complements $\mathscr{A}_n$, and consider a permutation $\pi = \langle \sigma_1, \ldots, \sigma_n \rangle$ of $\{1, \ldots, n\}$. Note that the word $w_\pi = (\sigma_1 \cdots \sigma_n \cdot \#)^\omega$ is not in L_n. Thus, w_π is accepted by $\mathscr{A}'_n$. Let r_π be an accepting run of $\mathscr{A}'_n$ on w_π, and let $S_\pi \subseteq Q'_n$ be the set of states that are visited infinitely often in r_π. We prove that for every two different permutations π_1 and π_2 of $\{1, \ldots, n\}$, it must be that $S_{\pi_1} \cap S_{\pi_2} = \emptyset$. Since there are $n!$ different permutations, this implies that $\mathscr{A}'_n$ must have at least $n!$ states.

Assume by way of contradiction that π_1 and π_2 are such that $S_{\pi_1} \cap S_{\pi_2} \neq \emptyset$. Let $q \in Q'_n$ be a state in $S_{\pi_1} \cap S_{\pi_2}$. We define three finite words in Σ_n^*:

- a prefix h of w_{π_1} with which r_{π_1} moves from an initial state of $\mathscr{A}'_n$ to q,
- an infix u_1 of w_{π_1} that includes the permutation π_1 and with which r_{π_1} moves from q back to q and visits α'_n at least once when it does so, and
- an infix u_2 of w_{π_2} that includes the permutation π_2 and with which r_{π_2} moves from q back to q.

Note that since $\mathscr{A}'_n$ accepts w_{π_1} and w_{π_2}, the words h, u_1, and u_2 exist. In particular, since r_{π_1} is accepting and q is visited infinitely often in r_{π_1}, there is at least one (in fact, there are infinitely many) infix in r_{π_1} that leads from q to itself and visits α'_n.

Consider the word $w = h \cdot (u_1 \cdot u_2)^\omega$. We claim that $w \in L_n$ and $w \in \mathscr{L}(\mathscr{A}'_n)$, contradicting the fact that $\mathscr{A}'_n$ complements $\mathscr{A}_n$. We first point to an accepting run r of $\mathscr{A}'_n$ on w. The run r first follows r_{π_1} and gets to q while reading h. Then, the run r repeatedly follows the run r_{π_1} when it moves from q via α'_n back to q while reading u_1, and the run r_{π_2} when it moves from q back to q while reading u_2. It is easy to see that r is a run on w that visits α'_n infinitely often, thus $w \in \mathscr{L}(\mathscr{A}'_n)$.

Now, let $\pi_1 = \langle \sigma^1_1, \ldots, \sigma^1_n \rangle$ and $\pi_2 = \langle \sigma^2_1, \ldots, \sigma^2_n \rangle$, and let j be the minimal index for which $\sigma^1_j \neq \sigma^2_j$. There must exist $j < k, l \leq n$ such that $\sigma^1_j = \sigma^2_k$, and $\sigma^2_j = \sigma^1_l$. Since u_1 includes the permutation π_1 and u_2 includes the permutation π_2, the pairs $\sigma^1_j\sigma^1_{j+1}$, $\sigma^1_{j+1}\sigma^1_{j+2}, \ldots, \sigma^1_{l-1}\sigma^1_l$, $\sigma^1_l\sigma^2_{j+1} (= \sigma^2_j\sigma^2_{j+1})$, $\sigma^2_{j+1}\sigma^2_{j+2}, \ldots, \sigma^2_{k-1}\sigma^2_k$, $\sigma^2_k\sigma^1_{j+1} (= \sigma^1_j\sigma^1_{j+1})$ repeat infinitely often. Hence, $w \in L_n$ and we are done. □

Remark 1 Note that the alphabets of the languages L_n used in the proof of Theorem 5 depend on n. As shown in [50], it is possible to encode the languages and prove a $2^{\Omega(n \log n)}$ lower bound with a fixed alphabet.

We note that the upper and lower bounds here are based on classical and relatively simple constructions and proofs, but are still not tight. A tighter upper bound, based on a restriction and a more precise counting of the required level rankings has been suggested in [18], and tightened further in [66]. An alternative approach, yielding a similar bound, is based on tracking levels of "split trees"—run trees in which only essential information about the history of each run is maintained [17, 30]. A tighter lower bound, based on the notion of full automata, is described in [80].

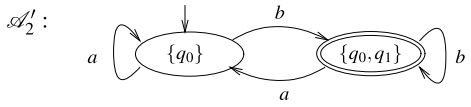


Fig. 5 The DBW obtained by applying the subset construction to $\mathscr{A}_2$

4.2.3 Determinization

Nondeterministic automata on finite words can be determinized by applying the subset construction [63]. Starting with a nondeterministic automaton $\mathscr{A} = \langle \Sigma, Q, Q_0, \delta, \alpha \rangle$, the subset construction generates a deterministic automaton $\mathscr{A}'$ with state space 2^Q. The intuition is that the single run of $\mathscr{A}'$ is in state $S \in 2^Q$ after reading a word $w \in \Sigma^*$ iff S is the set of states that $\mathscr{A}$ could have been at, in one of its runs, after reading w. Accordingly, the single initial state of $\mathscr{A}'$ is the set Q_0, and the transition of $\mathscr{A}'$ from a state $S \in 2^Q$ and a letter $\sigma \in \Sigma$ is the set $\bigcup_{s \in S} \delta(s, \sigma)$. Since $\mathscr{A}'$ accepts exactly all words on which there is a run of $\mathscr{A}$ that ends in α, the set of accepting states of $\mathscr{A}'$ consists of these sets S such that $S \cap \alpha \neq \emptyset$. The exponential blow-up that the subset construction involves is justified by a matching lower bound.

It is not hard to see that the subset construction does not result in an equivalent automaton when applied to an NBW. For example, applying the subset construction to the NBW $\mathscr{A}_2$ from Example 2 results in the DBW $\mathscr{A}_2'$ in Fig. 5. Recall that $\mathscr{A}_2$ recognizes the language of all words with finitely many a's. On the other hand, $\mathscr{A}_2'$ recognizes the language of all words with infinitely many b's. Thus, $\mathscr{L}(\mathscr{A}_2') \neq \mathscr{L}(\mathscr{A}_2)$. For example, the word $(a \cdot b)^\omega$ is in $\mathscr{L}(\mathscr{A}_2') \setminus \mathscr{L}(\mathscr{A}_2)$.

Note that not only $\mathscr{L}(\mathscr{A}_2') \neq \mathscr{L}(\mathscr{A}_2)$, there is no way to define a Büchi acceptance condition on top of the structure of $\mathscr{A}_2'$ and obtain a DBW that would be equivalent to $\mathscr{A}_2$. In fact, as we shall see now, there is no DBW that is equivalent to $\mathscr{A}_2$.

Theorem 6 ([49]) *There is a language L that is NBW-recognizable but not DBW-recognizable.*

Proof Consider the language L described in Example 2. I.e., L is over the alphabet $\{a, b\}$ and it consists of all infinite words in which a occurs only finitely many times. The language L is recognized by the NBW $\mathscr{A}_2$ appearing in Fig. 2. We prove that L is not DBW-recognizable. Assume by way of contradiction that $\mathscr{A}$ is a DBW such that $\mathscr{L}(\mathscr{A}) = L$. Let $\mathscr{A} = \langle \{a, b\}, Q, q_0, \delta, \alpha \rangle$. Recall that δ can be viewed as a partial mapping from $Q \times \{a, b\}^*$ to Q.

Consider the infinite word $w_0 = b^\omega$. Clearly, w_0 is in L, so the run of $\mathscr{A}$ on w_0 is accepting. Thus, there is $i_1 \geq 0$ such that the prefix b^{i_1} of w_0 is such that $\delta(q_0, b^{i_1}) \in \alpha$. Consider now the infinite word $w_1 = b^{i_1} \cdot a \cdot b^\omega$. Clearly, w_1 is also in L, so the run of $\mathscr{A}$ on w_1 is accepting. Thus, there is $i_2 \geq 0$ such that the prefix $b^{i_1} \cdot a \cdot b^{i_2}$ of w_1 is such that $\delta(q_0, b^{i_1} \cdot a \cdot b^{i_2}) \in \alpha$. In a similar fashion we can continue to find

indices $i_1, i_2, \ldots$ such that $\delta(q_0, b^{i_1} \cdot a \cdot b^{i_2} \cdot a \cdots ab^{i_j}) \in \alpha$ for all $j \geq 1$. Since Q is finite, there are iterations j and k, such that $1 \leq j < k \leq |\alpha| + 1$ and there is a state q such that $q = \delta(q_0, b^{i_1} \cdot a \cdot b^{i_2} \cdot a \cdots a \cdot b^{i_j}) = \delta(q_0, b^{i_1} \cdot a \cdot b^{i_2} \cdot a \cdots a \cdot b^{i_k})$. Since $j < k$, the extension $ab^{i_{j+1}} \cdots b^{i_{k-1}} \cdot a \cdot b^{i_k}$ is not empty and at least one state in α is visited when $\mathscr{A}$ loops in q while running through it. It follows that the run of $\mathscr{A}$ on the word

$$w = b^{i_1} \cdot a \cdot b^{i_2} \cdot a \cdots ab^{i_j} \cdot \left(ab^{i_{j+1}} \cdots b^{i_{k-1}} \cdot a \cdot b^{i_k}\right)^{\omega}$$

is accepting. But w has infinitely many occurrences of a, so it is not in L, and we have reached a contradiction. □

Note that the complementary language $(a+b)^{\omega} \setminus L$, which is the language of infinite words in which a occurs infinitely often, is recognized by the DBW described in Example 1. It follows that DBWs are not closed under complementation.

A good way to understand why the subset construction does not work for determinization on NBWs is to note that the DBW $\mathscr{A}_2'$ discussed above accepts exactly all words that have infinitely many prefixes on which there is a run of $\mathscr{A}_2$ that reaches an accepting state. Since $\mathscr{A}_2$ is nondeterministic, the different runs need not extend each other, and thus they need not induce a single run of $\mathscr{A}_2$ that visits the accepting state infinitely often. In Sect. 4.3.2, we are going to return to this example and study NBW determinization in general. Here, we use the "extend each other" intuition for the following characterization of languages that are DBW-recognizable.

For a language $R \subseteq \Sigma^*$, let $lim(R) \subseteq \Sigma^{\omega}$ be the set of infinite words that have infinitely many prefixes in R. Formally, $lim(R) = \{w = \sigma_1 \cdot \sigma_2 \cdots : \sigma_1 \cdots \sigma_i \in R \text{ for infinitely many } i \geq 0\}$. Thus, lim is an operator that takes a language of finite words and turns it into a language of infinite words. For example, if R is the language of words ending with a, then $lim(R)$ is the language of words with infinitely many a's.

Theorem 7 ([49]) *A language $L \subseteq \Sigma^{\omega}$ is DBW-recognizable iff there is a regular language $R \subseteq \Sigma^*$ such that $L = lim(R)$.*

Proof Assume first that L is DBW-recognizable. Let $\mathscr{A}$ be a DBW that recognizes L, let $\mathscr{A}_F$ be $\mathscr{A}$ when viewed as an automaton on finite words, and let $R = \mathscr{L}(\mathscr{A}_F)$. It is easy to see that since $\mathscr{A}$, and therefore also $\mathscr{A}_F$, are deterministic, we have that $\mathscr{L}(\mathscr{A}) = lim(R)$. Assume now that there is a regular language $R \subseteq \Sigma^*$ such that $L = lim(R)$. Let $\mathscr{A}$ be a deterministic automaton on finite words that recognizes R, and let $\mathscr{A}_B$ be $\mathscr{A}$ when viewed as a DBW. Again, since $\mathscr{A}$ is deterministic, and thus runs on different prefixes of a word extend each other, it is easy to see that $\mathscr{L}(\mathscr{A}_B) = lim(\mathscr{L}(\mathscr{A}))$. Hence, L is DBW-recognizable. □

Note that a DBW-recognizable language may be the limit of several different regular languages. As we demonstrate in Theorem 8 below, this explains why, unlike the case of automata on finite words, a language may have different minimal DBWs. In fact, while minimization of automata on finite words can be done in polynomial time, the problem of DBW minimization in NP-complete [68].

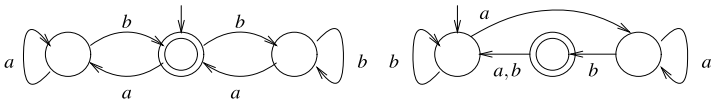


Fig. 6 Two minimal DBWs for L

Theorem 8 *A DBW-recognizable language L may not have a unique minimal DBW.*

Proof Let $\Sigma = \{a, b\}$. Consider the language L of all words that contain infinitely many a's and infinitely many b's. It is not hard to prove that L cannot be recognized by a DBW with two states. Figure 6 describes two three-state, and thus minimal, DBWs for the language. In fact, each of the states in the automata may be the initial state, so the figure describes six such (non-isomorphic) automata. □

Theorem 6 implies that we cannot hope to develop a determinization construction for NBWs. Suppose, however, that we have changed the definition of acceptance, and work with a definition in which a run is accepting iff it visits the set of accepting states only finitely often; i.e., $inf(r) \cap \alpha = \emptyset$. It is not hard to see that using such a definition, termed *co-Büchi*, we could have a deterministic automaton that recognizes the language L used in the proof of Theorem 6. In particular, the language is recognized by the deterministic automaton $\mathscr{A}_1$ from Fig. 1 when we view it as a co-Büchi automaton. While the co-Büchi condition enables us to recognize the language L with a deterministic automaton, it is not expressive enough to recognize all languages that are recognizable by NBWs. In Sect. 4.3, we are going to introduce and study several acceptance conditions, and see how NBWs can be determinized using acceptance conditions that are stronger than the Büchi and co-Büchi conditions.

4.3 Additional Acceptance Conditions

The Büchi acceptance condition suggests one possible way to refer to $inf(r)$ for defining when a run r is accepting. The fact that DBWs are strictly less expressive than NBWs motivates the introduction of other acceptance conditions. In this section we review some acceptance conditions and discuss the expressive power and succinctness of the corresponding automata.

Consider an automaton with state space Q. We define the following acceptance conditions.

- *Co-Büchi*, where $\alpha \subseteq Q$, and a run r is accepting iff $inf(r) \cap \alpha = \emptyset$.
- *Generalized Büchi*, where $\alpha = \{\alpha_1, \ldots, \alpha_k\}$, with $\alpha_i \subseteq Q$, and a run r is accepting if $inf(r) \cap \alpha_i \neq \emptyset$ for all $1 \leq i \leq k$.
- *Rabin*, where $\alpha = \{\langle \alpha_1, \beta_1 \rangle, \langle \alpha_2, \beta_2 \rangle, \ldots, \langle \alpha_k, \beta_k \rangle\}$, with $\alpha_i, \beta_i \subseteq Q$, and a run r is accepting if for some $1 \leq i \leq k$, we have that $inf(r) \cap \alpha_i \neq \emptyset$ and $inf(r) \cap \beta_i = \emptyset$.

- *Streett*, where $\alpha = \{\langle \alpha_1, \beta_1 \rangle, \langle \alpha_2, \beta_2 \rangle, \ldots, \langle \alpha_k, \beta_k \rangle\}$, with $\alpha_i, \beta_i \subseteq Q$ and a run r is accepting if for all $1 \leq i \leq k$, we have that $inf(r) \cap \alpha_i = \emptyset$ or $inf(r) \cap \beta_i \neq \emptyset$.
- *Parity*, where $\alpha = \{\alpha_1, \alpha_2, \ldots, \alpha_k\}$ with $\alpha_1 \subseteq \alpha_2 \subseteq \cdots \subseteq \alpha_k = Q$, and a run r is accepting if the minimal index i for which $inf(r) \cap \alpha_i \neq \emptyset$ is even.
- *Muller*, where $\alpha = \{\alpha_1, \alpha_2, \ldots, \alpha_k\}$, with $\alpha_i \subseteq Q$ and a run r is accepting if for some $1 \leq i \leq k$, we have that $inf(r) = \alpha_i$.

The number of sets in the generalized Büchi, parity, and Muller acceptance conditions or pairs in the Rabin and Streett acceptance conditions is called the *index* of the automaton. We extend our NBW and DBW notations to the above classes of automata, and we use the letters C, R, S, P, and M to denote co-Büchi, Rabin, Streett, parity, and Muller automata, respectively. Thus, for example, DPW stands for deterministic parity automata. We sometimes talk about satisfaction of an acceptance condition α by a set S of states. As expected, S satisfies α iff a run r with $inf(r) = S$ is accepting. For example, a set S satisfies a Büchi condition α iff $S \cap \alpha \neq \emptyset$.

It is easy to see that the co-Büchi acceptance condition is dual to the Büchi acceptance condition in the sense that a run r is accepting with a Büchi condition α iff r is not accepting when α is viewed as a co-Büchi condition, and vice versa. This implies, for example, that for a deterministic automaton $\mathscr{A}$, we have that $\mathscr{L}(\mathscr{A}_B) = \Sigma^\omega \setminus \mathscr{L}(\mathscr{A}_C)$, where $\mathscr{A}_B$ and $\mathscr{A}_C$ are the automata obtained by viewing $\mathscr{A}$ as a Büchi and co-Büchi automaton, respectively. Similarly, the Rabin acceptance condition is dual to the Streett acceptance condition. Indeed, if $\alpha = \{\langle \alpha_1, \beta_1 \rangle, \langle \alpha_2, \beta_2 \rangle, \ldots, \langle \alpha_k, \beta_k \rangle\}$, then for every run r, there is no $1 \leq i \leq k$ such that $inf(r) \cap \alpha_i \neq \emptyset$ and $inf(r) \cap \beta_i = \emptyset$ iff for all $1 \leq i \leq k$, we have that $inf(r) \cap \alpha_i = \emptyset$ or $inf(r) \cap \beta_i \neq \emptyset$.

For two classes γ and κ of automata, we say that γ is *at least as expressive as* κ if for every κ-automaton $\mathscr{A}$, there is a γ-automaton $\mathscr{A}'$ such that $\mathscr{L}(\mathscr{A}') = \mathscr{L}(\mathscr{A})$. If both γ is at least as expressive as κ and κ is at least as expressive as γ, then γ is *as expressive as* κ. One way to prove that γ is at least as expressive as κ is to show a translation of κ-automata to γ-automata. In the next section we are going to see such translations. As we shall see there, NBWs are as expressive as NRWs, NSWs, NPWs, and NMWs. On the other hand, NCWs are strictly less expressive than NBWs. Also, as we shall see in Sect. 4.3.2, nondeterminism does not add expressive power in automata with the richer acceptance conditions. Thus, DRWs, DSWs, DPWs, and DMWs recognize all ω-regular languages, and are as expressive as NBWs. This is in contrast with the Büchi condition, where, as we have seen in Theorem 6, NBWs are strictly more expressive than DBWs. Finally, nondeterminism does not add expressive power also in co-Büchi automata, thus NCWs are as expressive as DCW, where both are weaker than NBW and coincide with the set of languages whose complement languages are NBW-recognizable (see Remark 3).

4.3.1 Translations Among the Different Classes

We distinguish between three types of translations among automata of the different classes: (1) Translations among the different conditions. This is the simplest case, where it is possible to translate the acceptance condition itself, regardless of the automaton on top of which the condition is defined. For example, a Büchi acceptance condition α is equivalent to the Rabin condition $\{\langle \alpha, \emptyset \rangle\}$. (2) Translations in which we still do not change the structure of the automaton, yet the definition of the acceptance condition may depend on its structure. Following the terminology of [35], we refer to such translations as *typed*. (3) Translations that manipulate the state space. This is the most general case, where the translation may involve a blow-up in the state space of the automaton. Accordingly, here we are interested also in the *succinctness* of the different classes, namely the worst-case bound on the blow-up when we translate. In this section we survey the three types.

4.3.1.1 Translations Among the Different Conditions

Some conditions are special cases of other conditions, making the translation among the corresponding automata straightforward. We list these cases below. Consider an automaton with state space Q.

- A Büchi condition α is equivalent to the Rabin condition $\{\langle \alpha, \emptyset \rangle\}$, the Streett condition $\{\langle Q, \alpha \rangle\}$, and the parity condition $\{\emptyset, \alpha, Q\}$.
- A co-Büchi condition α is equivalent to the Rabin condition $\{\langle Q, \alpha \rangle\}$, the Streett condition $\{\langle \alpha, \emptyset \rangle\}$, and the parity condition $\{\alpha, Q\}$.
- A generalized Büchi condition $\{\alpha_1, \ldots, \alpha_k\}$ is equivalent to the Streett condition $\{\langle Q, \alpha_1 \rangle, \langle Q, \alpha_2 \rangle, \ldots, \langle Q, \alpha_k \rangle\}$.
- A parity condition $\{\alpha_1, \ldots, \alpha_k\}$ (for simplicity, assume that k is even; otherwise, we can duplicate α_k) is equivalent to the Rabin condition $\{\langle \alpha_2, \alpha_1 \rangle, \langle \alpha_4, \alpha_3 \rangle, \ldots, \langle \alpha_k, \alpha_{k-1} \rangle\}$, and to the Streett condition $\{\langle \alpha_1, \emptyset \rangle, \langle \alpha_3, \alpha_2 \rangle, \ldots, \langle \alpha_{k-1}, \alpha_{k-2} \rangle\}$. (Recall that $\alpha_k = Q$, so there is no need to include the pair $\langle Q, \alpha_k \rangle$ in the Streett condition.)
- A Büchi, co-Büchi, Rabin, Streett, or parity acceptance condition α is equivalent to the Muller condition $\{F : F \text{ satisfies } \alpha\}$.

4.3.1.2 Typeness

In [35], the authors studied the expressive power of DBWs and introduced the notion of typeness for automata. For two classes γ and κ of automata, we say that γ is κ-type if for every γ-automaton $\mathcal{A}$, if $\mathcal{L}(\mathcal{A})$ is κ-recognizable, then it is possible to define a κ-automaton $\mathcal{A}'$ such that $\mathcal{L}(\mathcal{A}') = \mathcal{L}(\mathcal{A})$ and $\mathcal{A}'$ differs from $\mathcal{A}$ only in the definition of the acceptance condition. Clearly, if an acceptance condition can be translated to another acceptance condition, as discussed in Sect. 4.3.1.1, then typeness for the corresponding classes follows. Interestingly, typeness may be valid also when γ is more expressive than κ. We demonstrate this below.

Theorem 9 ([35]) *DRWs are DBW-type.*

Proof Consider a DRW $\mathscr{A} = \langle \Sigma, Q, q_0, \delta, \alpha \rangle$. Let $\alpha = \{\langle \alpha_1, \beta_1 \rangle, \ldots, \langle \alpha_k, \beta_k \rangle\}$. We say that a state $q \in Q$ is good in $\mathscr{A}$ if all the cycles $C \subseteq Q$ that contain q satisfy the acceptance condition α. Consider the DBW $\mathscr{A}' = \langle \Sigma, Q, q_0, \delta, \alpha' \rangle$, where $\alpha' = \{q : q \text{ is good in } \mathscr{A}\}$. We prove that if $\mathscr{A}$ is DBW-recognizable, then $\mathscr{L}(\mathscr{A}) = \mathscr{L}(\mathscr{A}')$. Hence, if $\mathscr{A}$ is DBW-recognizable, then there is a DBW equivalent to $\mathscr{A}$ that can be obtained by only changing the acceptance condition of $\mathscr{A}$.

We first prove that $\mathscr{L}(\mathscr{A}') \subseteq \mathscr{L}(\mathscr{A})$. In fact, this direction is independent of $\mathscr{A}$ being DBW-recognizable. Consider a word $w \in \mathscr{L}(\mathscr{A}')$. Let r be the accepting run of $\mathscr{A}'$ on w. Since r is accepting, there is a state $q \in inf(r) \cap \alpha'$. Recall that the states in $inf(r)$ constitute an SCS and thus also constitute a cycle that contains q. Therefore, as q is good, $inf(r)$ satisfies α, and r is also an accepting run of $\mathscr{A}$ on w. Hence, $w \in \mathscr{L}(\mathscr{A})$ and we are done.

We now prove that $\mathscr{L}(\mathscr{A}) \subseteq \mathscr{L}(\mathscr{A}')$. Consider a word $w \in \mathscr{L}(\mathscr{A})$. Let r be the accepting run of $\mathscr{A}$ on w. We prove that $inf(r) \cap \alpha' \neq \emptyset$. Assume by way of contradiction that $inf(r) \cap \alpha' = \emptyset$. Thus, no state in $inf(r)$ is good, so for each state $q \in inf(r)$, there is a cycle C_q that contains q and does not satisfy α. By [49], a deterministic automaton $\mathscr{A}$ recognizes a language that is in DBW iff for every strongly connected component C of $\mathscr{A}$, if C satisfies α, then all the strongly connected components C' with $C' \supseteq C$ satisfy α too. Consider the strongly connected component $C' = \bigcup_{q \in inf(r)} C_q$. Since C' contains $inf(r)$, and $inf(r)$ satisfies α, then, by the above, C' satisfies α too. Therefore, there is $1 \leq i \leq k$ such that $C' \cap \alpha_i \neq \emptyset$ and $C' \cap \beta_i = \emptyset$. Consider a state $s \in C' \cap \alpha_i$. Let q be such that $s \in C_q$. Observe that $C_q \cap \alpha_i \neq \emptyset$ and $C_q \cap \beta_i = \emptyset$, contradicting the fact that C_q does not satisfy α. □

Theorem 10 ([35]) *DSWs are not DBW-type.*

Proof Consider the automaton $\mathscr{A}_1$ appearing in Fig. 1, now with the Streett condition $\{\langle \{q_0, q_1\}, \{q_0\} \rangle, \langle \{q_0, q_1\}, \{q_1\} \rangle\}$. The language L of $\mathscr{A}_1$ then consists of exactly all words with infinitely many a's and infinitely many b's. As we have seen in the proof of Theorem 8, L is DBW-recognizable. Yet, none of the four possibilities to define a DBW on top of the structure of $\mathscr{A}_1$ result in a DBW that recognizes L. □

Note that, by dualization, we get from Theorems 9 and 10 that DSWs are DCW-type and DRWs are not DCW-type.

The definition of typeness may be applied to nondeterministic automata too. As we show below, typeness need not coincide for nondeterministic and deterministic automata.

Theorem 11 ([38]) *DBWs are DCW-type, but NBWs are not NCW-type.*

Proof The first claim follows from the fact that DBWs are a special case of DSWs, and the latter are DCW-type. For the second claim, consider the NBW $\mathscr{A}$ appearing in Fig. 7. The NBW $\mathscr{A}$ has two initial states, in two disjoint components. Thus, the

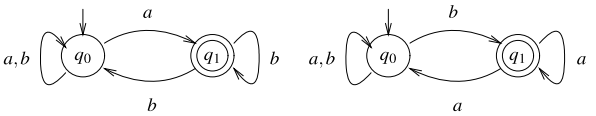


Fig. 7 An NBW that recognizes an NCW-recognizable language but has no equivalent NCW on the same structure

language of $\mathscr{A}$ is the union of the languages of the two NBWs associated with its two components. The NBW on the left accepts all words over the alphabet $\{a, b\}$ that satisfy "eventually a and infinitely many b's". The NBW on the right accepts all words that satisfy "eventually b and infinitely many a's". While each of these languages is not NCW-recognizable, their union recognizes the language L of all words satisfying "eventually a and eventually b", which is NCW-recognizable. It is not hard to see that none of the four possibilities to define a co-Büchi acceptance condition on top of $\mathscr{A}$ result in an NCW that recognizes L. □

Researchers have considered additional variants of typeness. We mention two here. Let γ and κ be two acceptance conditions. In *powerset typeness*, we ask whether a deterministic κ-automaton can be defined on top of the subset construction of a nondeterministic γ-automaton. For example, NCWs are DBW-powerset-type: if the language of an NCW $\mathscr{A}$ is DBW-recognizable, then a DBW for $\mathscr{L}(\mathscr{A})$ can be defined on top of the subset construction of $\mathscr{A}$ [51]. In *combined typeness*, we ask whether the ability to define a certain language on top of the same automaton using two different acceptance conditions implies we can define it using a third, weaker, condition. For example, DRWs+DSWs are DPW-type: if a language L can be defined on top of a deterministic automaton $\mathscr{A}$ using both a Streett and a Rabin acceptance condition, then L can be defined on top of $\mathscr{A}$ also using a parity acceptance condition [4, 81]. For more results on typeness, see [38].

4.3.1.3 Translations That Require a New State Space

We now turn to the most general type of translations—those that may involve a blow-up in the state space. We do not specify all the translations, and rather describe the translation of nondeterministic generalized Büchi, Rabin, and Streett automata into NBWs. For the case of NSW, where the translation involves a blow-up that is exponential in the index, we also describe a lower bound.

Theorem 12 *Let $\mathscr{A}$ be a nondeterministic generalized Büchi automaton with n states and index k. There is an NBW $\mathscr{A}'$ with $n \cdot k$ states such that $\mathscr{L}(\mathscr{A}') = \mathscr{L}(\mathscr{A})$.*

Proof Let $\mathscr{A} = \langle \Sigma, Q, Q_0, \delta, \{\alpha_1, \ldots, \alpha_k\}\rangle$. The idea of the construction of $\mathscr{A}'$ is similar to the one used for defining the intersection of NBWs. Informally, $\mathscr{A}'$ consists of k copies of $\mathscr{A}$, and it stays in the i-th copy until it visits a state in α_i, in which case it moves to the next copy (modulo k). The acceptance condition of $\mathscr{A}'$ then makes sure that all copies are visited infinitely often.

Formally, $\mathscr{A}' = \langle \Sigma, Q', Q'_0, \delta', \alpha \rangle$, where

- $Q' = Q \times \{1, \ldots, k\}$.
- $Q'_0 = Q \times \{1\}$.
- For every $q \in Q$, $i \in \{1, \ldots, k\}$, and $\sigma \in \Sigma$, we have $\delta(\langle q, i \rangle, \sigma) = \delta(q, \sigma) \times \{j\}$, where $j = i$ if $q \notin \alpha_i$ and $j = (i \bmod k) + 1$ if $q \in \alpha_i$.
- $\alpha = \alpha_1 \times \{1\}$. Note that after a visit to α_1 in the first copy, the run moves to the second copy, where it waits for visits to α_2, and so on until it visits α_k in the k-th copy, and moves back to the first copy. Therefore, infinitely many visits in α_1 in the first copy indeed ensure that all copies, and thus also all α_i's are visited infinitely often. □

Theorem 13 *Let $\mathscr{A}$ be an NRW with n states and index k. There is an NBW $\mathscr{A}'$ with at most $n(k+1)$ states such that $\mathscr{L}(\mathscr{A}') = \mathscr{L}(\mathscr{A})$.*

Proof Let $\mathscr{A} = \langle \Sigma, Q, Q_0, \delta, \{\langle \alpha_1, \beta_1 \rangle, \ldots, \langle \alpha_k, \beta_k \rangle\} \rangle$. It is easy to see that $\mathscr{L}(\mathscr{A}) = \bigcup_{i=1}^{k} \mathscr{L}(\mathscr{A}_i)$, where $\mathscr{A}_i = (\Sigma, Q, Q_0, \delta, \{\langle \alpha_i, \beta_i \rangle\})$. By Theorem 1, NBWs are closed under union. It therefore suffices to show a translation to NBWs of NRWs with index 1.

Consider an NRW $\mathscr{U} = \langle \Sigma, Q, Q_0, \delta, \{\langle \alpha, \beta \rangle\} \rangle$ with index 1. We translate $\mathscr{U}$ to an NBW $\mathscr{U}'$. The idea of the construction is similar to the one used for complementing DBWs: the NBW $\mathscr{U}'$ consists of two copies of $\mathscr{U}$, and it nondeterministically moves to the second copy, which contains only states that are not in β, and in which it has to visit infinitely many states in α. Formally, $\mathscr{U}' = \langle \Sigma, Q', Q'_0, \delta', \alpha' \rangle$, where

- $Q' = (Q \times \{0\}) \cup ((Q \setminus \beta) \times \{1\})$.
- $Q'_0 = Q_0 \times \{0\}$.
- For all $q \in Q$ and $\sigma \in \Sigma$, we have $\delta'(\langle q, 0 \rangle, \sigma) = (\delta(q, \sigma) \times \{0\}) \cup ((\delta(q, \sigma) \setminus \beta) \times \{1\})$, and $\delta'(\langle q, 1 \rangle, \sigma) = (\delta(q, \sigma) \setminus \beta) \times \{1\}$ for $q \in Q \setminus \beta$.
- $\alpha' = \alpha \times \{1\}$.

Since for an NRW $\mathscr{U}$ with n states, the NBW $\mathscr{U}'$ has at most $2n$ states, the union NBW has at most $2nk$ states. Now, in order to reduce the state space to $n(k+1)$, we observe that the first copy of $\mathscr{U}'$ can be shared by all $\mathscr{A}_i$'s. Thus, $\mathscr{A}'$ guesses both the pair α_i, β_i with which the acceptance condition is satisfied and the point from which states from β_i are no longer visited. Formally, we define $\mathscr{A}' = \langle \Sigma, Q', Q_0 \times \{0\}, \delta', \alpha' \rangle$, where

- $Q' = (Q \times \{0\}) \cup \bigcup_{1 \le i \le k} ((Q \setminus \beta_i) \times \{i\})$.
- For all $q \in Q$ and $\sigma \in \Sigma$, we have $\delta'(\langle q, 0 \rangle, \sigma) = (\delta(q, \sigma) \times \{0\}) \cup \bigcup_{1 \le i \le k} ((\delta(q, \sigma) \setminus \beta_i) \times \{i\})$, and $\delta'(\langle q, i \rangle, \sigma) = (\delta(q, \sigma) \setminus \beta_i) \times \{i\}$ for $1 \le i \le k$ and $q \in Q \setminus \beta_i$.
- $\alpha' = \bigcup_{1 \le i \le k} \alpha_i \times \{i\}$. □

Translating NRWs to NBWs, we took the union of the NRWs of index 1 that are obtained by decomposing the acceptance condition. For NSW, it is tempting to proceed dually, and define the NBW as the intersection of the NSWs of index 1

that are obtained by decomposing the acceptance condition. Such an intersection, however, may accept words that are not in the language of the NSW. To see this, consider an automaton $\mathscr{A}$, a Streett acceptance condition $\alpha = \{\langle \alpha_1, \beta_1 \rangle, \langle \alpha_2, \beta_2 \rangle\}$, and a word w. It may be that there is a run r_1 of $\mathscr{A}$ on w that satisfies the Streett acceptance condition $\{\langle \alpha_1, \beta_1 \rangle\}$ and also a run r_2 of $\mathscr{A}$ on w that satisfies the Streett acceptance condition $\{\langle \alpha_2, \beta_2 \rangle\}$. Yet, the runs r_1 and r_2 may be different, and there need not be a run of $\mathscr{A}$ on w that satisfies both $\{\langle \alpha_1, \beta_1 \rangle\}$ and $\{\langle \alpha_2, \beta_2 \rangle\}$. Consequently, the translation of NSWs to NBWs has to consider the relation among the different pairs in α, giving rise to a blow-up that is exponential in k. Formally, we have the following.

Theorem 14 *Let $\mathscr{A}$ be an NSW with n states and index k. There is an NBW $\mathscr{A}'$ with at most $n(1 + k2^k)$ states such that $\mathscr{L}(\mathscr{A}') = \mathscr{L}(\mathscr{A})$.*

Proof Let $\mathscr{A} = \langle \Sigma, Q, Q_0, \delta, \{\langle \alpha_1, \beta_1 \rangle, \ldots, \langle \alpha_k, \beta_k \rangle\} \rangle$. Recall that in an accepting run r of $\mathscr{A}$, we have that $inf(r) \cap \alpha_i = \emptyset$ or $inf(r) \cap \beta_i \neq \emptyset$ for all $1 \leq i \leq k$. For $I \subseteq \{1, \ldots, k\}$, we define an NBW $\mathscr{A}_I$ that accepts exactly all words w such that there is a run r of $\mathscr{A}$ on w for which $inf(r) \cap \alpha_i = \emptyset$ for all $i \in I$ and $inf(r) \cap \beta_i \neq \emptyset$ for all $i \notin I$. Thus, I indicates how the acceptance condition α is satisfied. It is easy to see that $\mathscr{L}(\mathscr{A}) = \bigcup_{I \subseteq \{1,\ldots,k\}} \mathscr{L}(\mathscr{A}_I)$.

The idea behind the construction of $\mathscr{A}_I$ is similar to the "two copies" idea we have seen above, except that now, in the copy in which $\mathscr{A}_I$ avoids the states in α_i, for all $i \in I$, it also has to visit all the states in β_i, for $i \notin I$. This can be easily achieved by first defining $\mathscr{A}_I$ as a nondeterministic generalized Büchi automaton. Formally, we define $\mathscr{A}_I = \langle \Sigma, Q_I, Q'_0, \delta_I, \beta_I \rangle$ as follows. Let $\alpha_I = \bigcup_{i \in I} \alpha_i$. Then,

- $Q_I = (Q \times \{0\}) \cup ((Q \setminus \alpha_I) \times \{1\})$.
- $Q'_0 = Q_0 \times \{0\}$.
- For every $q \in Q$ and $\sigma \in \Sigma$, we have $\delta_I(\langle q, 0 \rangle, \sigma) = (\delta(q, \sigma) \times \{0\}) \cup ((\delta(q, \sigma) \setminus \alpha_I) \times \{1\})$. For $q \in Q \setminus \alpha_I$, we also have $\delta_I(\langle q, 1 \rangle, \sigma) = (\delta(q, \sigma) \setminus \alpha_I) \times \{1\}$.
- $\beta_I = \{\beta_i \times \{1\} : i \notin I\}$.

Since $\mathscr{A}_I$ has at most $2n$ states and index k, an equivalent Büchi automaton has at most $2nk$ states. A slightly more careful analysis observes that the generalized Büchi condition applies only to the second copy of $\mathscr{A}_I$, thus a translation to NBW results in an automaton with at most $n + nk$ states. The automaton $\mathscr{A}'$ is then the union of all the 2^k NBWs obtained from the different $\mathscr{A}_I$ and thus has at most $(n + nk)2^k$ states. Moreover, as in the proof of Theorem 13, the first copy of all the NBWs in the union can be shared, tightening the bound further to $n + nk2^k$. □

In Theorem 15 below we show that the exponential blow-up in the translation of NSWs to NBWs cannot be avoided. In fact, as the theorem shows, the blow-up may occur even when one starts with a DSW.

Theorem 15 ([65]) *There is a family of languages $L_1, L_2, \ldots$ such that L_n can be recognized by a DSW with $3n$ states and index $2n$, but an NBW for L_n has at least 2^n states.*

Proof Let $\Sigma = \{0, 1, 2\}$. We can view an infinite word over Σ as a word $w \in (\Sigma^n)^\omega$, thus $w = u_1 \cdot u_2 \cdot u_3 \cdots$, where each u_j is a word in Σ^n. We refer to such words as *blocks*. We say that index $i \in \{0, \ldots, n-1\}$ is 0-active in w iff there are infinitely many j's such that the i-th letter in u_j is 0. Similarly, i is 1-active in w iff there are infinitely many j's such that the i-th letter in u_j is 1. For $n \geq 1$, let

$$L_n = \{w : \text{ for all } 0 \leq i \leq n-1, \text{ the index } i \text{ is 0-active in } w \text{ iff } i \text{ is 1-active in } w\}.$$

We first describe a DSW $\mathscr{A}_n$ with $3n$ states such that $\mathscr{L}(\mathscr{A}_n) = L_n$. We define $\mathscr{A}_n = \langle \{0, 1, 2\}, Q_n, \{\langle 0, 0\rangle\}, \delta_n, \alpha_n \rangle$, where

- $Q_n = \{1, \ldots, n\} \times \{0, 1, 2\}$. Intuitively, $\mathscr{A}_n$ moves to the state $\langle i, \sigma \rangle$ after it reads the $(i-1)$-th letter in the current block, and this letter is σ. Accordingly, an index $0 \leq i \leq n-1$ is σ-active in w iff the run of $\mathscr{A}_n$ on w visits states in $\{i+1\} \times \{\sigma\}$ infinitely often.
- For all $0 \leq i \leq n-1$ and $\sigma, \sigma' \in \{0, 1, 2\}$, we have $\delta_n(\langle i, \sigma \rangle, \sigma') = \langle (i+1) \bmod n, \sigma' \rangle$.
- $\alpha_n = \bigcup_{1 \leq i \leq n} \{\langle \{\langle i, 0\rangle\}, \{\langle i, 1\rangle\} \rangle, \langle \{\langle i, 1\rangle\}, \{\langle i, 0\rangle\} \rangle\}$.

It is easy to see that $\mathscr{A}_n$ has $3n$ states and that $\mathscr{L}(\mathscr{A}_n) = L_n$. Now, assume by way of contradiction that there is an NBW $\mathscr{A}'_n$ that recognizes L_n and has fewer than 2^n states. We say that a position in a word or in a run of $\mathscr{A}'_n$ is *relevant* if it is 0 mod n. That is, $\mathscr{A}'_n$ starts to read each block in a relevant position. For a set $S \subseteq \{0, \ldots, n-1\}$, let $w_S^0 \in \{0, 2\}^n$ be the word of length n in which for all $0 \leq i \leq n-1$, the i-th letter is 0 iff $i \in S$, and is 2 otherwise. Similarly, let $w_S^1 \in \{1, 2\}^n$ be the word in which the i-th letter is 1 iff $i \in S$, and is 2 otherwise. Note that if w_S^0 appears in a word w in infinitely many relevant positions, then all the indices in S are 0-active, and similarly for w_S^1 and 1-active. Consider the infinite word $w_S = ((w_S^0)^{2^n} \cdot w_S^1)^\omega$. Clearly, for index $i \in \{0, \ldots, n-1\}$, we have that i is 0-active in w_S iff i is 1-active in w_S iff $i \in S$. Hence, $w_S \in L_n$. Let r_S be an accepting run of $\mathscr{A}'_n$ on w_S. We say that a position $p \geq 0$ in r_S is *important* if it is relevant and there is a state q and a position $p' > p$ such that q is visited in both positions p and p' and the subword read between them is in $(w_S^0)^*$. We then say that q *supports* p. Let Q_S be the set of states that support infinitely many important positions. Since Q is finite and there are infinitely many relevant positions, the set Q_S is not empty. Since $\mathscr{A}'_n$ has fewer than 2^n states, there must be two subsets S and T, such that $T \neq S$ and $Q_S \cap Q_T \neq \emptyset$. Let S and T be two such subsets. Assume without loss of generality that $T \setminus S \neq \emptyset$, and let q be a state in $Q_S \cap Q_T$. By the definition of Q_T, there is $i \geq 1$ such that $\mathscr{A}'_n$ can move from q back to itself when it reads $(w_T^0)^i$. We claim that we can then obtain from w_S a word w'_S that is not in L_n and is accepted by $\mathscr{A}'_n$. We obtain w'_S by inserting the word $(w_T^0)^i$ inside the $(w_S^0)^{2^n}$ subwords whenever the run of $\mathscr{A}'_n$ reaches the state q in important positions. The accepting run of $\mathscr{A}'_n$ is then similar to r_S, except that we pump visits to q in important positions to traverse the cycle along which $(w_T^0)^i$ is read. Since $\mathscr{A}'_n$ is a Büchi automaton, the run stays accepting, whereas the word it reads has indices (those in $T \setminus S$) that are 0-active but not 1-active, and is therefore not in L'_n. □

4.3.2 Determinization of NBWs

Recall that NBWs are strictly more expressive than DBWs. In this section we describe the intuition behind a determinization construction that translates a given NBW to an equivalent DPW. Detailed description of the construction can be found in [59, 64, 67]. As in the case of NBW complementation, efforts to determinize NBWs started in the 1960s, and involve several landmarks. In [52], McNaughton proved that NBWs can be determinized and described a doubly-exponential translation of NBWs to DMWs. Only in 1988, Safra improved the bound and described an optimal translation of NBWs to DRWs: given an NBW with n states, the equivalent DRW has $2^{O(n \log n)}$ states and index n. A different construction, with similar bounds, was given in [58]. The same considerations that hold for NBW complementation can be used in order to show a matching $2^{\Omega(n \log n)}$ lower bound [50, 54]. While Safra's determinization construction is asymptotically optimal, efforts to improve it have continued, aiming at reducing the state space and generating automata with the parity acceptance condition. Indeed, the parity acceptance condition has important advantages: it is easy to complement, and when used as a winning condition in a two-player game, both players can proceed according to memoryless strategies. Also, solving parity games is easier than solving Rabin games [12, 29] (see Chap. 27). In [59], Piterman described a direct translation of NBW to DPW, which also reduces the state blow-up in Safra's determinization. Piterman's construction has been further tightened in [67]. The translation is a variant of Safra's determinization construction, and we present the intuition behind it here. It is important to note that in addition to efforts to improve Safra's determinization construction, there have been efforts to develop algorithms that avoid determinization in constructions and methodologies that traditionally involve determinization; e.g., complementation of NBW [44], LTL synthesis [45], and more [36].

Before we describe the intuition behind the determinization construction, let us understand why NBW determinization is a difficult problem. Consider the NBW $\mathscr{A}_2$ from Example 2. In Fig. 5 we described the DBW $\mathscr{A}_2'$ obtained by applying the subset construction to $\mathscr{A}_2$. While $\mathscr{A}_2$ recognizes the language of all words with finitely many a's, the DBW $\mathscr{A}_2'$ recognizes the language of all words with infinitely many b's. Why does the subset construction work for finite words and fail here? Consider the word $w = (b \cdot a)^\omega$. The fact the run of $\mathscr{A}_2'$ on w visits the state $\{q_0, q_1\}$ infinitely often implies that there are infinitely many prefixes of w such that $\mathscr{A}_2$ has a run on the prefix that ends in q_1. Nothing, however, is guaranteed about our ability to compose the runs on these prefixes into a single run. In particular, in the case of w, the run on each of the prefixes visits q_1 only once, as the destination of its last transition, and there is no way to continue and read the suffix of w from q_1.

Consider an NBW $\mathscr{A} = \langle \Sigma, Q, Q_0, \delta, \alpha \rangle$ and an input word $w = \sigma_1 \cdot \sigma_2 \cdot \sigma_3 \cdots$. As the example above demonstrates, an equivalent deterministic automaton should not only make sure that w has infinitely many prefixes on which $\mathscr{A}$ can reach α, but also that $\mathscr{A}$ does so with runs that can be composed into a single run. Let $S_0, S_1, S_2, \ldots \in (2^Q)^\omega$ be the result of applying the subset construction of $\mathscr{A}$

on w. That is, S_i is the set of states that $\mathscr{A}$ can be at after reading $\sigma_1 \cdot \sigma_2 \cdots \sigma_i$. The deterministic automaton that is equivalent to $\mathscr{A}$ tries to find a sequence $\tau = T_0, T_1, T_2, \ldots \in (2^Q)^\omega$ such that $T_0 \subseteq S_0$ and for all $i \geq 0$, we have that $T_{i+1} \subseteq \delta(T_i, \sigma_{i+1})$. In addition, there are infinitely many positions $j_1, j_2, j_3, \ldots$ such that for all $k \geq 1$, each of the states in $T_{j_{k+1}}$ is reachable from some state in T_{j_k} via a run that visits α. We refer to τ as a *witness sequence* and refer to the positions $j_1, j_2, j_3, \ldots$ as *break-points*. Note that for all $i \geq 0$ we have $T_i \subseteq S_i$, and that indeed $\mathscr{A}$ accepts w iff such a witness sequence exists. First, if $\mathscr{A}$ accepts w with a run $q_0, q_1, \ldots$, then we can take $T_i = \{q_i\}$. Also, if τ exists, then we can generate an accepting run of $\mathscr{A}$ on w by reaching some state in T_{j_1}, then reaching, via α, some state in T_{j_2}, then reaching, via α, some state in T_{j_3}, and so on. The big challenge in the determinization construction is to detect a witness sequence without guessing.

One naive way to detect a witness sequence is to maintain full information about the runs of $\mathscr{A}$ on w. In Sect. 4.2.2.2, we defined the run DAG G that embodies all the possible runs of $\mathscr{A}$ on w. The prefix of G up to level i clearly contains all the information one needs about S_i and the history of all the states in it. The prefixes of G, however, are of increasing and unbounded sizes. A key point in the determinization construction is to extract from each prefix of G a finite presentation that is sufficiently informative. For the case of finite words, this is easy—the set of states in the last level of the prefix (that is, S_i) is sufficient. For the case of infinite words, the presentation is much more complicated, and is based on the data structure of *history trees*.

Essentially, the history tree that is reached after reading a prefix of length i of w maintains subsets of S_i that may serve as T_i. One challenge is to maintain these subsets in a compact way. A second challenge is to use the parity acceptance condition in order to guarantee that one of the maintained subsets can indeed serve as T_i in a witness sequence. The first challenge is addressed by arranging all candidate subsets in a tree in which each state in S_i is associated with at most one node of the tree. This bounds the number of history trees by $n^{O(n)}$. The second challenge is addressed by updating the history trees in each transition in a way that relates the choice of the subset that would serve as T_i with the choice of the even index that witnesses the satisfaction of the parity condition: the subsets are ordered, essentially, according to their seniority—the point at which the deterministic automaton started to take them into account as a possible T_i. In each update, each subset may be declared as "stable", meaning that it continues to serve as a possible T_i, and may also be declared as "accepting", meaning that the position i is a break-point in the witness sequence in which T_i is a member. The parity acceptance condition then uses labels of seniority in order to look for a subset that is eventually always stable and infinitely often accepting. The above is only a high-level intuition, and in particular it misses the way in which the subsets are ordered and how the updates interfere with this order. As pointed out above, details can be found in the original papers [59, 64, 67].

4.4 Decision Procedures

Automata define languages, which are sets of words. Natural questions to ask about sets are whether they are trivial (that is, empty or universal), and whether two sets contain each other. Note that equivalence between two sets amounts to containment in both directions. In this section we study the following three problems, which address the above questions for languages defined by automata.

- The *non-emptiness* problem is to decide, given an automaton $\mathscr{A}$, whether $\mathscr{L}(\mathscr{A}) \neq \emptyset$.
- The *non-universality* problemis to decide, given an automaton $\mathscr{A}$, whether $\mathscr{L}(\mathscr{A}) \neq \Sigma^\omega$.
- The *language-containment* problem is to decide, given automata $\mathscr{A}_1$ and $\mathscr{A}_2$, whether $\mathscr{L}(\mathscr{A}_1) \subseteq \mathscr{L}(\mathscr{A}_2)$.

It is not hard to see that the non-emptiness and non-universality problems are dual, in the sense that an automaton is non-empty iff its complement is non-universal, and that both can be viewed as a special case of the language-containment problem. Indeed, if $\mathscr{A}_\bot$ and $\mathscr{A}_\top$ are such that $\mathscr{L}(\mathscr{A}_\bot) = \emptyset$ and $\mathscr{L}(\mathscr{A}_\top) = \Sigma^\omega$, then an automaton $\mathscr{A}$ is empty if $\mathscr{L}(\mathscr{A}) \subseteq \mathscr{L}(\mathscr{A}_\bot)$ and is universal if $\mathscr{L}(\mathscr{A}_\top) \subseteq \mathscr{L}(\mathscr{A})$. As we shall see below, however, the non-emptiness problem is easier than the other two. We note that the hardness results and proofs described in this section hold already for automata on finite words. We still present direct proofs for NBWs. An alternative would be to carry out a reduction from the setting of finite words.

Theorem 16 ([14, 15, 77]) *The non-emptiness problem for NBWs is decidable in linear time and is NLOGSPACE-complete.*

Proof Consider an NBW $\mathscr{A} = \langle \Sigma, Q, Q_0, \delta, \alpha \rangle$. Recall that $\mathscr{A}$ induces a directed graph $G_\mathscr{A} = \langle Q, E \rangle$ where $\langle q, q' \rangle \in E$ iff there is a letter σ such that $q' \in \delta(q, \sigma)$. We claim that $\mathscr{L}(\mathscr{A})$ is non-empty iff there are states $q_0 \in Q_0$ and $q_{acc} \in \alpha$ such that $G_\mathscr{A}$ contains a path leading from q_0 to q_{acc} and a cycle going though q_{acc}. Assume first that $\mathscr{L}(\mathscr{A})$ is non-empty. Then, there is an accepting run $r = q_0, q_1, \ldots$ of $\mathscr{A}$ on some input word, which corresponds to an infinite path of $G_\mathscr{A}$. Since r is accepting, some state $q_{acc} \in \alpha$ occurs in r infinitely often; in particular, there are i, j, where $0 \leq i < j$, such that $q_{acc} = q_i = q_j$. Thus, $q_0, \ldots, q_i$ corresponds to a (possibly empty) path from q_0 to q_{acc}, and $q_i, \ldots, q_j$ to a cycle going through q_{acc}.

Conversely, assume that $G_\mathscr{A}$ contains a path leading from q_0 to a state $q_{acc} \in \alpha$ and a cycle going though q_{acc}. We can then construct an infinite path of $G_\mathscr{A}$ starting at q_0 and visiting q_{acc} infinitely often. This path induces a run on a word accepted by $\mathscr{A}$.

Thus, NBW non-emptiness is reducible to graph reachability. The algorithm that proves membership in NLOGSPACE first guesses states $q_0 \in Q_0$ and $q_{acc} \in \alpha$, and then checks the reachability requirements by guessing a path from q_0 to q_{acc} and a path from q_{acc} to itself. Guessing these paths is done by remembering the current state on the path and the value of a counter for the length of the path traversed so far,

and proceeding to a successor state while increasing the counter. When the counter value exceeds $|Q|$, the algorithm returns "no" (that is, the guess is not good). Note that the algorithm has to remember q_0, q_{acc}, the current state and counter value, each requiring logarithmic space.

NLOGSPACE-hardness can be proved by an easy reduction from the reachability problem in directed graphs [28]. There, one is given a directed graph $G = \langle V, E\rangle$ along with two vertices s and t, and the goal is to decide whether there is a path from s to t. It is easy to see that such a path exists iff the NBW $\mathscr{A}_G = \langle \{a\}, V, \{s\}, \delta, \{t\}\rangle$ with $v' \in \delta(v, a)$ iff $E(v, v')$ or $v' = v = t$ is not empty.

To check non-emptiness in linear time, we first find the decomposition of $G_{\mathscr{A}}$ into SCCs [10, 73]. An SCC is nontrivial if it contains an edge, which means, since it is strongly connected, that it contains a cycle. It is not hard to see that $\mathscr{A}$ is non-empty iff from an SCC whose intersection with Q_0 is not empty it is possible to reach a nontrivial SCC whose intersection with α is not empty. □

Theorem 17 ([70]) *The non-universality problem for NBWs is decidable in exponential time and is PSPACE-complete.*

Proof Consider an NBW $\mathscr{A}$. Clearly, $\mathscr{L}(\mathscr{A}) \neq \Sigma^\omega$ iff $\Sigma^\omega \setminus \mathscr{L}(\mathscr{A}) \neq \emptyset$, which holds iff $\mathscr{L}(\mathscr{A}') \neq \emptyset$, where $\mathscr{A}'$ is an NBW that complements $\mathscr{A}$. Thus, to test $\mathscr{A}$ for non-universality, it suffices to test $\mathscr{A}'$ for non-emptiness. The construction of $\mathscr{A}'$ can proceed "on-the-fly" (that is, there is no need to construct and store $\mathscr{A}'$ and then perform the non-emptiness test, but rather it is possible to construct only the components required for the non-emptiness test on demand; such a construction requires only polynomial space). Hence, as $\mathscr{A}'$ is exponentially bigger than $\mathscr{A}$, the time and space bounds from Theorem 16 imply the two upper bounds.

For the lower bound, we do a reduction from polynomial-space Turing machines. The reduction does not use the fact that Büchi automata run on infinite words and follows the same considerations as the reduction showing that the non-universality problem is PSPACE-hard for nondeterministic automata on finite words [53]. Note that we could also have reduced from this latter problem, but preferred to give the details of the generic reduction.

Given a Turing machine T of space complexity $s(n)$, we construct an NBW $\mathscr{A}_T$ of size linear in T and $s(n)$ such that $\mathscr{A}_T$ is universal iff T does not accept the empty tape. We assume, without loss of generality, that all the computations of T eventually reach a final state. Also, once T reaches a final state it loops there forever. The NBW $\mathscr{A}_T$ accepts a word w iff w is not an encoding of a legal computation of T over the empty tape or if w is an encoding of a legal yet rejecting computation of T over the empty tape. Thus, $\mathscr{A}_T$ rejects a word w iff w encodes a legal and accepting computation of T over the empty tape. Hence, $\mathscr{A}_T$ is universal iff T does not accept the empty tape.

We now give the details of the construction of $\mathscr{A}_T$. Let $T = \langle \Gamma, Q, \rightarrow, q_0, q_{acc}, q_{req}\rangle$, where Γ is the alphabet, Q is the set of states, $\rightarrow \subseteq Q \times \Gamma \times Q \times \Gamma \times \{L, R\}$ is the transition relation (we use $(q, a) \rightarrow (q', b, \Delta)$ to indicate that when T is in state q and it reads the input a in the current tape cell, it moves to

state q', writes b in the current tape cell, and its reading head moves one cell to the left/right, according to Δ), and q_0, q_{acc}, and q_{rej} are the initial, accepting, and rejecting states.

We encode a configuration of T by a word $\#\gamma_1\gamma_2 \ldots (q, \gamma_i) \ldots \gamma_{s(n)}$. That is, a configuration starts with #, and all its other letters are in Γ, except for one letter in $Q \times \Gamma$. The meaning of such a configuration is that the j-th cell in T, for $1 \leq j \leq s(n)$, is labeled γ_j, the reading head points at cell i, and T is in state q. For example, the initial configuration of T is $\#(q_0, b)b \ldots b$ (with $s(n) - 1$ occurrences of b) where b stands for an empty cell. We can now encode a computation of T by a sequence of configurations.

Let $\Sigma = \{\#\} \cup \Gamma \cup (Q \times \Gamma)$ and let $\#\sigma_1 \ldots \sigma_{s(n)}\#\sigma'_1 \ldots \sigma'_{s(n)}$ be two successive configurations of T. We also set σ_0, σ'_0, and $\sigma_{s(n)+1}$ to #. For each triple $\langle \sigma_{i-1}, \sigma_i, \sigma_{i+1} \rangle$ with $1 \leq i \leq s(n)$, we know, by the transition relation of T, what σ'_i should be. In addition, the letter # should repeat exactly every $s(n) + 1$ letters. Let $next(\langle \sigma_{i-1}, \sigma_i, \sigma_{i+1} \rangle)$ denote our expectation for σ'_i. That is,

- $next(\langle \gamma_{i-1}, \gamma_i, \gamma_{i+1} \rangle) = next(\langle \#, \gamma_i, \gamma_{i+1} \rangle) = next(\langle \gamma_{i-1}, \gamma_i, \# \rangle) = \gamma_i$.
- $next(\langle (q, \gamma_{i-1}), \gamma_i, \gamma_{i+1} \rangle) = next(\langle (q, \gamma_{i-1}), \gamma_i, \# \rangle) =$

$$\begin{cases} \gamma_i & \text{if } (q, \gamma_{i-1}) \to (q', \gamma'_{i-1}, L) \\ (q', \gamma_i) & \text{if } (q, \gamma_{i-1}) \to (q', \gamma'_{i-1}, R) \end{cases}$$

- $next(\langle \gamma_{i-1}, (q, \gamma_i), \gamma_{i+1} \rangle) = next(\langle \#, (q, \gamma_i), \gamma_{i+1} \rangle) = next(\langle \gamma_{i-1}, (q, \gamma_i), \# \rangle) = \gamma'_i$ where $(q, \gamma_i) \to (q', \gamma'_i, \Delta)$.[3]
- $next(\langle \gamma_{i-1}, \gamma_i, (q, \gamma_{i+1}) \rangle) = next(\langle \#, \gamma_i, (q, \gamma_{i+1}) \rangle) =$

$$\begin{cases} \gamma_i & \text{if } (q, \gamma_{i+1}) \to (q', \gamma'_{i+1}, R) \\ (q', \gamma_i) & \text{if } (q, \gamma_{i+1}) \to (q', \gamma'_i, L) \end{cases}$$

- $next(\langle \sigma_{s(n)}, \#, \sigma'_1 \rangle) = \#$.

Consistency with *next* now gives us a necessary condition for a trace to encode a legal computation. In addition, the computation should start with the initial configuration.

In order to check consistency with *next*, the NBW $\mathcal{A}_T$ can use its nondeterminism and guess when there is a violation of *next*. Thus, $\mathcal{A}_T$ guesses $\langle \sigma_{i-1}, \sigma_i, \sigma_{i+1} \rangle \in \Sigma^3$, guesses a position in the trace, checks whether the three letters to be read starting in this position are σ_{i-1}, σ_i, and σ_{i+1}, and checks whether $next(\langle \sigma_{i-1}, \sigma_i, \sigma_{i+1} \rangle)$ is not the letter to come $s(n) + 1$ letters later. Once $\mathcal{A}_T$ sees such a violation, it goes to an accepting sink. In order to check that the first configuration is not the initial configuration, $\mathcal{A}_T$ simply compares the first $s(n) + 1$ letters with $\#(q_0, b)b \ldots b$. Finally, checking whether a legal computation is rejecting is also easy: the computation should reach a configuration in which T visits q_{rej}. □

[3] We assume that the reading head of T does not "fall" from the right or the left boundaries of the tape. Thus, the case where $(i = 1)$ and $(q, \gamma_i) \to (q', \gamma'_i, L)$ and the dual case where $(i = s(n))$ and $(q, \gamma_i) \to (q', \gamma'_i, R)$ are not possible.

Theorem 18 ([70]) *The containment problem for NBWs is decidable in exponential time and is PSPACE-complete.*

Proof Consider NBWs $\mathscr{A}_1$ and $\mathscr{A}_2$. Note that $\mathscr{L}(\mathscr{A}_1) \subseteq \mathscr{L}(\mathscr{A}_2)$ iff $\mathscr{L}(\mathscr{A}_1) \cap (\Sigma^\omega \setminus \mathscr{L}(\mathscr{A}_2)) = \emptyset$, which holds iff $\mathscr{L}(\mathscr{A}') = \emptyset$, where $\mathscr{A}'$ is an NBW for the intersection of $\mathscr{A}_1$ with an NBW that complements $\mathscr{A}_2$. Thus, to check the containment of $\mathscr{A}_1$ in $\mathscr{A}_2$ we can test $\mathscr{A}'$ for emptiness. Since the construction of $\mathscr{A}'$ can proceed on-the-fly and its size is linear in the size of $\mathscr{A}_1$ and exponential in the size of $\mathscr{A}_2$, the required complexity follows, as in the proof of Theorem 17. Since $\mathscr{A}_2$ is universal iff $\Sigma^\omega \subseteq \mathscr{L}(\mathscr{A}_2)$ and Σ^ω can be recognized by an NBW with one state, hardness in PSPACE follows from hardness of the universality problem. □

Recall that the algorithm for deciding non-emptiness of an NBW $\mathscr{A}$ operates on the graph $G_\mathscr{A}$ induced by $\mathscr{A}$ and thus ignores the alphabet of $\mathscr{A}$. In particular, the algorithm does not distinguish between deterministic and nondeterministic automata. In contrast, the algorithms for deciding universality and containment complement the NBW and thus, can benefit from determinization.

Theorem 19 *The non-emptiness, non-universality, and containment problems for DBWs are NLOGSPACE-complete.*

Proof When applied to DBWs, the intermediate automaton $\mathscr{A}'$ used in the proofs of Theorems 17 and 18 is polynomial in the size of the input, thus its non-emptiness can be tested in NLOGSPACE. Hardness in NLOGSPACE follows from the fact that reachability in directed graphs can be reduced to the three problems. □

Theorems 16, 17, and 18 refer to Büchi automata. For the other types of automata, one can translate to NBWs and apply the algorithm for them. While in many cases this results in an optimal algorithm, sometimes it is more efficient to work directly on the input automaton. In particular, for NSW, the translation to NBW results in an NBW with $O(n^2 2^k)$ states, whereas non-emptiness can be checked in subquadratic time [16, 25]. We note, however, that unlike NBWs and NRWs, for which the non-emptiness problem is NLOGSPACE-complete, it is PTIME-complete for NSWs [42]. For NPWs, the translation to NBWs results in an NBW with $O(nk)$ states, whereas non-emptiness can be checked in time $O(n \log k)$ [32].

4.5 Alternating Automata on Infinite Words

In [7], Chandra et al. introduced alternating Turing machines. In the alternating model, the states of the machine, and accordingly also its configurations, are partitioned into existential and universal ones. When the machine is in an existential configuration, one of its successors should lead to acceptance. When the machine is in a universal configuration, all its successors should lead to acceptance. In this section we define alternating Büchi automata [55] and study their properties.

4.5.1 Definition

For a given set X, let $\mathscr{B}^+(X)$ be the set of positive Boolean formulas over X (i.e., Boolean formulas built from elements in X using $\wedge$ and $\vee$), where we also allow the formulas ***true*** and ***false***. For $Y \subseteq X$, we say that Y *satisfies* a formula $\theta \in \mathscr{B}^+(X)$ iff the truth assignment that assigns *true* to the members of Y and assigns *false* to the members of $X \setminus Y$ satisfies θ. We say that Y satisfies θ *in a minimal manner* if no strict subset of Y satisfies θ. For example, the sets $\{q_1, q_3\}$, $\{q_2, q_3\}$, and $\{q_1, q_2, q_3\}$ all satisfy the formula $(q_1 \vee q_2) \wedge q_3$, yet only the first two sets satisfy it in a minimal manner. Also, the set $\{q_1, q_2\}$ does not satisfy this formula.

Consider an automaton $\mathscr{A} = \langle \Sigma, Q, Q_0, \delta, \alpha \rangle$. We can represent δ using $\mathscr{B}^+(Q)$. For example, a transition $\delta(q, \sigma) = \{q_1, q_2, q_3\}$ of a nondeterministic automaton $\mathscr{A}$ can be written as $\delta(q, \sigma) = q_1 \vee q_2 \vee q_3$. The dual of nondeterminism is universality. A word w is accepted by a universal automaton $\mathscr{A}$ if all the runs of $\mathscr{A}$ on w are accepting. Accordingly, if $\mathscr{A}$ is universal, then the transition can be written as $\delta(q, \sigma) = q_1 \wedge q_2 \wedge q_3$. While transitions of nondeterministic and universal automata correspond to disjunctions and conjunctions, respectively, transitions of alternating automata can be arbitrary formulas in $\mathscr{B}^+(Q)$. We can have, for instance, a transition $\delta(q, \sigma) = (q_1 \wedge q_2) \vee (q_3 \wedge q_4)$, meaning that the automaton accepts a word of the form $\sigma \cdot w$ from state q, if it accepts w from both q_1 and q_2 or from both q_3 and q_4. Such a transition combines existential and universal choices.

Formally, an *alternating automaton on infinite words* is a tuple $\mathscr{A} = \langle \Sigma, Q, q_0, \delta, \alpha \rangle$, where Σ, Q, and α are as in nondeterministic automata, $q_0 \in Q$ is an initial state (we will later explain why it is technically easier to assume a single initial state), and $\delta : Q \times \Sigma \to \mathscr{B}^+(Q)$ is a transition function. In order to define runs of alternating automata, we first have to define trees and labeled trees. A *tree* is a prefix-closed set $T \subseteq \mathbb{N}^*$ (i.e., if $x \cdot c \in T$, where $x \in \mathbb{N}^*$ and $c \in \mathbb{N}$, then also $x \in T$). The elements of T are called *nodes*, and the empty word ε is the *root* of T. For every $x \in T$, the nodes $x \cdot c$, for $c \in \mathbb{N}$, are the *successors* of x. A node is a *leaf* if it has no successors. We sometimes refer to the length $|x|$ of x as its *level* in the tree. A *path* π of a tree T is a set $\pi \subseteq T$ such that $\varepsilon \in \pi$ and for every $x \in \pi$, either x is a leaf or there exists a unique $c \in \mathbb{N}$ such that $x \cdot c \in \pi$. Given an alphabet Σ, a *Σ-labeled tree* is a pair $\langle T, V \rangle$ where T is a tree and $V : T \to \Sigma$ maps each node of T to a letter in Σ.

While a run of a nondeterministic automaton on an infinite word is an infinite sequence of states, a run of an alternating automaton is a Q-labeled tree. Formally, given an infinite word $w = \sigma_1 \cdot \sigma_2 \cdots$, a run of $\mathscr{A}$ on w is a Q-labeled tree $\langle T_r, r \rangle$ such that the following hold:

- $\varepsilon \in T_r$ and $r(\varepsilon) = q_0$.
- Let $x \in T_r$ with $r(x) = q$ and $\delta(q, \sigma_{|x|+1}) = \theta$. There is a (possibly empty) set $S = \{q_1, \ldots, q_k\}$ such that S satisfies θ in a minimal manner and for all $1 \leq c \leq k$, we have that $x \cdot c \in T_r$ and $r(x \cdot c) = q_c$.

For example, if $\delta(q_0, \sigma_1) = (q_1 \vee q_2) \wedge (q_3 \vee q_4)$, then possible runs of $\mathscr{A}$ on w have a root labeled q_0, have one node in level 1 labeled q_1 or q_2, and have another node

in level 1 labeled q_3 or q_4. Note that if $\theta = \textbf{\textit{true}}$, then x does not have children. This is the reason why T_r may have leaves. Also, since there exists no set S satisfying θ for $\theta = \textbf{\textit{false}}$, we cannot have a run that takes a transition with $\theta = \textbf{\textit{false}}$.

A run $\langle T_r, r\rangle$ is *accepting* iff all its infinite paths, which are labeled by words in Q^ω, satisfy the acceptance condition. A word w is accepted iff there exists an accepting run on it. Note that while conjunctions in the transition function of $\mathscr{A}$ are reflected in branches of $\langle T_r, r\rangle$, disjunctions are reflected in the fact we can have many runs on the same word. The language of $\mathscr{A}$, denoted $\mathscr{L}(\mathscr{A})$, is the set of infinite words that $\mathscr{A}$ accepts. We use ABW to abbreviate alternating Büchi word automata.

Example 4 For $n \geq 1$, let $\Sigma_n = \{1, 2, \ldots, n\}$. We describe an ABW $\mathscr{A}_n$ such that $\mathscr{A}_n$ accepts exactly all words $w \in \Sigma_n^\omega$ such that w contains the subword $i \cdot i \cdot i$ for all letters $i \in \Sigma_n$.

We define $\mathscr{A}_n = \langle \Sigma_n, Q_n, q_0, \delta, \emptyset\rangle$, where

- $Q_n = \{q_0\} \cup (\Sigma \times \{3, 2, 1\})$. Thus, in addition to an initial state, $\mathscr{A}_n$ contains three states for each letter $i \in \Sigma_n$, where state $\langle i, c\rangle$, for $c \in \{1, 2, 3\}$, waits for a subword i^c.
- In its first transition, $\mathscr{A}_n$ spawns into n copies, with copy i waiting for the subword i^3 (or i^2, in case the first letter read is i). Thus, for all $i \in \Sigma_n$, we have $\delta_n(q_0, i) = \langle i, 2\rangle \wedge \bigwedge_{j \neq i} \langle j, 3\rangle$. In addition, for all $i \in \Sigma_n$ and $c \in \{3, 2, 1\}$, we have

$$\delta_n(\langle i, c\rangle, j) = \begin{cases} \langle i, c-1\rangle & \text{if } j = i \text{ and } c \in \{3, 2\}, \\ \textbf{\textit{true}} & \text{if } j = i \text{ and } c = 1, \\ \langle i, 3\rangle & \text{if } j \neq i. \end{cases}$$

Note that no state in Q_n is accepting. Thus, all copies have to eventually take the transition to $\textbf{\textit{true}}$, guaranteeing that $i \cdot i \cdot i$ is indeed read, for all $i \in \Sigma_n$. Note also that while $\mathscr{A}_n$ has $3n + 1$ states, it is not hard to prove that an NBW for the language is exponential in n, as it has to remember the subsets of letters for which the subword $i \cdot i \cdot i$ has already appeared.

A slightly more general definition of alternating automata could replace the single initial state by an *initial transition* in $\mathscr{B}^+(Q)$, describing possible subsets of states from which the word should be accepted. Staying with the definition of a set of initial states used in nondeterministic automata would have broken the symmetry between the existential and universal components of alternation.

4.5.2 Closure Properties

The rich structure of alternating automata makes it easy to define the union and intersection of ABWs. Indeed, the same way union is easy for automata with nondeterminism, intersection is easy for automata with universal branches.

Theorem 20 *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be ABWs with n_1 and n_2 states, respectively. There are ABWs $\mathcal{A}_\cup$ and $\mathcal{A}_\cap$ such that $\mathcal{L}(\mathcal{A}_\cup) = \mathcal{L}(\mathcal{A}_1) \cup \mathcal{L}(\mathcal{A}_2)$, $\mathcal{L}(\mathcal{A}_\cap) = \mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$, and $\mathcal{A}_\cup$ and $\mathcal{A}_\cap$ have $n_1 + n_2 + 1$ states.*

Proof Let $\mathcal{A}_1 = \langle \Sigma, Q_1, q_1^0, \delta_1, \alpha_1 \rangle$ and $\mathcal{A}_2 = \langle \Sigma, Q_2, q_2^0, \delta_2, \alpha_2 \rangle$. We assume, without loss of generality, that Q_1 and Q_2 are disjoint. We define $\mathcal{A}_\cup$ as the union of $\mathcal{A}_1$ and $\mathcal{A}_2$, with an additional initial state that proceeds like the union of the initial states of $\mathcal{A}_1$ and $\mathcal{A}_2$. Thus, $\mathcal{A}_\cup = \langle \Sigma, Q_1 \cup Q_2 \cup \{q_0\}, q_0, \delta, \alpha_1 \cup \alpha_2 \rangle$, where $\delta(q_0, \sigma) = \delta_1(q_1^0, \sigma) \vee \delta_2(q_2^0, \sigma)$, and for every state $q \in Q_1 \cup Q_2$, we have that $\delta(q, \sigma) = \delta_i(q, \sigma)$, for the index $i \in \{1, 2\}$ such that $q \in Q_i$. It is easy to see that for every word $w \in \Sigma^\omega$, the ABW $\mathcal{A}$ has an accepting run on w iff at least one of the ABWs $\mathcal{A}_1$ and $\mathcal{A}_2$ has an accepting run on w. The definition of $\mathcal{A}_\cap$ is similar, except that from q_0 we proceed with the conjunction of the transitions from q_1^0 and q_2^0. □

We note that with a definition of ABWs in which an initial transition in $\mathcal{B}^+(Q)$ is allowed, closing ABWs under union and intersection can be done by applying the corresponding operation on the initial transitions. We proceed to closure of ABWs under complementation. Given a transition function δ, let $\tilde{\delta}$ denote the function dual to δ. That is, for every q and σ with $\delta(q, \sigma) = \theta$, we have that $\tilde{\delta}(q, \sigma) = \tilde{\theta}$, where $\tilde{\theta}$ is obtained from θ by switching $\vee$ and $\wedge$ and switching ***true*** and ***false***. If, for example, $\theta = p \vee (\textbf{\textit{true}} \wedge q)$, then $\tilde{\theta} = p \wedge (\textbf{\textit{false}} \vee q)$. Given an acceptance condition α, let $\tilde{\alpha}$ be an acceptance condition that dualizes α. Thus, a set of states S satisfies α iff S does not satisfy $\tilde{\alpha}$. In particular, if α is a Büchi condition, then $\tilde{\alpha}$ is a co-Büchi condition. For deterministic automata, it is easy to complement an automaton $\mathcal{A}$ by dualizing the acceptance condition. In particular, given a DBW $\mathcal{A}$, viewing $\mathcal{A}$ as a DCW complements its language. For an NBW $\mathcal{A}$, the situation is more involved as we have to make sure that all runs satisfy the dual condition. This can be done by viewing $\mathcal{A}$ as a universal co-Büchi automaton. As Lemma 3 below argues, this approach can be generalized to all alternating automata and acceptance conditions.

Lemma 3 ([58]) *Given an alternating automaton $\mathcal{A} = \langle \Sigma, Q, q_0, \delta, \alpha \rangle$, the alternating automaton $\tilde{\mathcal{A}} = \langle \Sigma, Q, q_0, \tilde{\delta}, \tilde{\alpha} \rangle$ is such that $\mathcal{L}(\tilde{\mathcal{A}}) = \Sigma^\omega \setminus \mathcal{L}(\mathcal{A})$.*

Lemma 3 suggests a straightforward translation of an ABW $\mathcal{A}$ to a complementing ACW $\tilde{\mathcal{A}}$, and vice versa. In order to end up with an ABW, one has to translate $\tilde{\mathcal{A}}$ to an ABW [44], which uses the ranking method described in the context of NBW complementation and involves a quadratic blow-up:

Theorem 21 ([44]) *Given an ABW $\mathcal{A}$ with n states, there is an ABW $\mathcal{A}'$ with $O(n^2)$ states such that $\mathcal{L}(\mathcal{A}') = \Sigma^\omega \setminus \mathcal{L}(\mathcal{A})$.*

We note that the ABW constructed in the proof of Theorem 21 is a *weak alternating automaton* [56]. In a weak automaton, each SCC of the automaton is either contained in α or is disjoint from α. Every infinite path of a run ultimately gets "trapped" within some SCC. The path then satisfies the acceptance condition iff this

component is contained in α. It is easy to see that weak automata are a special case of both Büchi and co-Büchi alternating automata. A run gets trapped in a component contained in α iff it visits α infinitely often iff it visits $Q \setminus \alpha$ only finitely often. The study of weak alternating automata is motivated by the fact that the translation of formulas in several temporal logics to alternating automata results in weak automata [46, 56]. Another motivation is the fact that dualizing a weak automaton is straightforward: taking $\tilde{\alpha} = Q \setminus \alpha$ amounts to switching the classification of accepting and rejecting sets, and thus dualizes the acceptance condition.

Remark 2 In the non-elementary translation of monadic second-order logic formulas to NBWs [6], an exponential blow-up occurs with each negation. While a blow-up that is non-elementary in the quantifier alternation depth is unavoidable, the fact that complementation is easy for alternating automata raises the question whether ABWs may be used in a simpler decision procedure. The negative answer follows from the fact that the *existential projection* operator, which is easy for nondeterministic automata, involves an exponential blow-up when applied to alternating automata. For a language $L \subseteq (\Sigma_1 \times \Sigma_2)^\omega$, we define the existential projection of L on Σ_1 as the language L_1 of all words $w_1 \in \Sigma_1^\omega$ such that there is a word $w_2 \in \Sigma_2^\omega$ for which $w_1 \otimes w_2 \in L$, where $w_1 \otimes w_2$ is the word over $\Sigma_1 \times \Sigma_2$ obtained by "merging" the letters of w_1 and w_2 in the expected way. For example, $abba \otimes 0010 = \langle a0\rangle\langle b0\rangle\langle b1\rangle\langle a0\rangle$. Given an NBW for L, it is easy to see that an NBW for L_1 can be obtained by replacing a letter $\langle \sigma_1, \sigma_2\rangle$ by the letter σ_1. Such a simple replacement, however, would not work for alternating automata. Indeed, there, one has to ensure that different copies of the automaton proceed according to the same word over Σ_2. Consequently, existential projection requires alternation removal. In the context of translations of formulas to automata, the exponential blow-up with each negation when working with NBWs is traded for an exponential blow-up with each existential quantifier when working with ABWs. It is easy to see, say by pushing negations inside, that negations and existential quantifiers can be traded also at the syntactic level of the formula.

4.5.3 Decision Procedures

The rich structure of alternating automata makes them exponentially more succinct than nondeterministic automata. On the other hand, reasoning about alternating automata is complicated. For example, while the algorithm for testing the non-emptiness of a nondeterministic automaton can ignore the alphabet and be reduced to reachability questions in the underlying graph of the automaton, ignoring the alphabet in an alternating automaton leads to an algorithm with a one-sided error. Indeed, as noted in the context of existential projection in Remark 2, the algorithm should make sure that the different copies it spawns into follow the same word. Consequently, many algorithms for alternating automata involve alternation removal—a translation to an equivalent nondeterministic automaton. Below we describe such a translation for the case of Büchi automata.

Theorem 22 ([55]) *Consider an ABW $\mathcal{A}$ with n states. There is an NBW $\mathcal{A}'$ with 3^n states such that $\mathcal{L}(\mathcal{A}') = \mathcal{L}(\mathcal{A})$.*

Proof The automaton $\mathcal{A}'$ guesses a run of $\mathcal{A}$. At a given point of a run of $\mathcal{A}'$, it keeps in its memory the states in a whole level of the run tree of $\mathcal{A}$. As it reads the next input letter, it guesses the states in the next level of the run tree of $\mathcal{A}$. In order to make sure that every infinite path visits states in α infinitely often, $\mathcal{A}'$ keeps track of states that "owe" a visit to α. Let $\mathcal{A} = \langle \Sigma, Q, q_0, \delta, \alpha \rangle$. Then $\mathcal{A}' = \langle \Sigma, 2^Q \times 2^Q, \langle \{q_0\}, \emptyset \rangle, \delta', 2^Q \times \{\emptyset\} \rangle$, where δ' is defined, for all $\langle S, O \rangle \in 2^Q \times 2^Q$ and $\sigma \in \Sigma$, as follows.

- If $O \neq \emptyset$, then $\delta'(\langle S, O \rangle, \sigma) = \{\langle S', O' \setminus \alpha \rangle \ : \ S' \text{ satisfies } \bigwedge_{q \in S} \delta(q, \sigma), O' \subseteq S', \text{ and } O' \text{ satisfies } \bigwedge_{q \in O} \delta(q, \sigma)\}$.
- If $O = \emptyset$, then $\delta'(\langle S, O \rangle, \sigma) = \{\langle S', S' \setminus \alpha \rangle \ : \ S' \text{ satisfies } \wedge_{q \in S} \delta(q, \sigma)\}$.

Note that all the reachable states $\langle S, O \rangle$ in $\mathcal{A}'$ satisfy $O \subseteq S$. Accordingly, if the number of states in $\mathcal{A}$ is n, then the number of states in $\mathcal{A}'$ is at most 3^n. □

Note that the construction has the flavor of the subset construction [63], but in a dual interpretation: a set of states is interpreted conjunctively: the suffix of the word has to be accepted from all the states in S. While such a dual subset construction is sufficient for automata on finite words, the case of Büchi requires also the maintenance of a subset O of S, leading to a $3^{O(n)}$, rather than a $2^{O(n)}$, blow-up. As shown in [3], this additional blow-up cannot be avoided.

Remark 3 It is not hard to see that if $\mathcal{A}$ is a universal automaton (that is, the transition function δ only has conjunctions), then the automaton $\mathcal{A}'$ constructed in the proof of Theorem 22 is deterministic. Indeed, in the definition of $\delta'(\langle S, O \rangle, \sigma)$, there is a single set S' that satisfies $\wedge_{q \in S} \delta(q, \sigma)$ in a minimal manner. It follows that universal Büchi automata are not more expressive than DBWs. Dually, NCWs are not more expressive than DCW: Given an NCW $\mathcal{A}$, we can apply the construction above on the dual universal Büchi automaton $\tilde{\mathcal{A}}$ (see Lemma 3), and then dualize the obtained DBW. We end up with a DCW equivalent to $\mathcal{A}$.

We can now use alternation removal in order to solve decision problems for alternating automata.

Theorem 23 *The non-emptiness, non-universality, and containment problems for ABW are PSPACE-complete.*

Proof We describe the proof for the non-emptiness problem. Since ABWs are easily closed for negation and intersection, the proof for non-universality and containment is similar. Consider an ABW $\mathcal{A}$. In order to check $\mathcal{A}$ for non-emptiness, we translate it into an NBW $\mathcal{A}'$ and check the non-emptiness of $\mathcal{A}'$. By Theorem 22, the size of $\mathcal{A}'$ is exponential in the size of $\mathcal{A}$. Since the construction of $\mathcal{A}'$ can proceed

on-the-fly, and, by Theorem 16, its non-emptiness can be checked in NLOGSPACE, membership in PSPACE follows.

In order to prove hardness in PSPACE, we do a reduction from NBW non-universality. Given an NBW $\mathcal{A}$, we have that $\mathcal{L}(\mathcal{A}) \neq \Sigma^\omega$ iff $\Sigma^\omega \setminus \mathcal{L}(\mathcal{A}) \neq \emptyset$. Thus, non-universality of $\mathcal{A}$ can be reduced to non-emptiness of an automaton $\mathcal{A}'$ that complements $\mathcal{A}$. Since we can define $\mathcal{A}'$ as an ABW with quadratically many states, hardness in PSPACE follows. □

4.6 Automata-Based Algorithms

In this section we describe the application of automata theory in formal verification. Recall that the logic LTL is used for specifying properties of reactive systems. The syntax and semantics of LTL are described in Chap. 2. For completeness, we describe them here briefly. Formulas of LTL are constructed from a set AP of atomic propositions using the usual Boolean operators and the temporal operators X ("next time") and U ("until"). Formally, an LTL formula over AP is defined as follows:

- ***true***, ***false***, or p, for $p \in AP$.
- $\neg\psi_1$, $\psi_1 \wedge \psi_2$, $X\psi_1$, or $\psi_1 U \psi_2$, where ψ_1 and ψ_2 are LTL formulas.

The semantics of LTL is defined with respect to infinite computations $\pi = \sigma_1, \sigma_2, \sigma_3, \ldots$, where for every $j \geq 1$, the set $\sigma_j \subseteq AP$ is the set of atomic propositions that hold in the j-th position of π. Systems that generate computations are modeled by *Kripke structures*. A (finite) Kripke structure is a tuple $K = \langle AP, W, W_0, R, \ell \rangle$, where AP is a finite set of atomic propositions, W is a finite set of states, $W_0 \subseteq W$ is a set of initial states, $R \subseteq W \times W$ is a transition relation, and $\ell : W \to 2^{AP}$ maps each state w to the set of atomic propositions that hold in w. We require that each state has at least one successor. That is, for each state $w \in W$ there is at least one state w' such that $R(w, w')$. A path in K is an infinite sequence $\rho = w_0, w_1, w_2, \ldots$ of states such that $w_0 \in W_0$ and for all $i \geq 0$, we have $R(w_i, w_{i+1})$. The path ρ induces the computation $\ell(w_0), \ell(w_1), \ell(w_2), \ldots$.

Consider a computation $\pi = \sigma_1, \sigma_2, \sigma_3, \ldots$. We denote the suffix $\sigma_j, \sigma_{j+1}, \ldots$ of π by π^j. We use $\pi \models \psi$ to indicate that an LTL formula ψ holds in the computation π. The relation $\models$ is inductively defined as follows:

- For all π, we have that $\pi \models$ ***true*** and $\pi \not\models$ ***false***.
- For an atomic proposition $p \in AP$, we have that $\pi \models p$ iff $p \in \sigma_1$.
- $\pi \models \neg\psi_1$ iff $\pi \not\models \psi_1$.
- $\pi \models \psi_1 \wedge \psi_2$ iff $\pi \models \psi_1$ and $\pi \models \psi_2$.
- $\pi \models X\psi_1$ iff $\pi^2 \models \psi_1$.
- $\pi \models \psi_1 U \psi_2$ iff there exists $k \geq 1$ such that $\pi^k \models \psi_2$ and $\pi^i \models \psi_1$ for all $1 \leq i < k$.

Writing LTL formulas, it is convenient to use the abbreviations G ("always"), F ("eventually"), and R ("release"). Formally, the abbreviations follow the following semantics.

- $F\psi_1 = \mathbf{true}U\psi_1$. That is, $\pi \models F\psi_1$ iff there exists $k \geq 1$ such that $\pi^k \models \psi_1$.
- $G\psi_1 = \neg F\neg\psi_1$. That is, $\pi \models G\psi_1$ iff for all $k \geq 1$ we have that $\pi^k \models \psi_1$.
- $\psi_1 R\psi_2 = \neg((\neg\psi_1)U(\neg\psi_2))$. That is, $\pi \models \psi_1 R\psi_2$ iff for all $k \geq 1$, if $\pi^k \not\models \psi_2$, then there is $1 \leq i < k$ such that $\pi^i \models \psi_1$.

Each LTL formula ψ over AP defines a language $\mathcal{L}(\psi) \subseteq (2^{AP})^\omega$ of the computations that satisfy ψ, Formally,

$$\mathcal{L}(\psi) = \left\{\pi \in \left(2^{AP}\right)^\omega \,:\, \pi \models \psi\right\}.$$

Two natural problems arise in the context of systems and their specifications:

- *Satisfiability:* given an LTL formula ψ, is there a computation π such that $\pi \models \psi$?
- *Model Checking:* given a Kripke structure K and an LTL formula ψ, do all the computations of K satisfy ψ?

We describe a translation of LTL formulas into Büchi automata and discuss how such a translation is used for solving the above two problems.

4.6.1 Translating LTL to Büchi Automata

In this section we describe a translation of LTL formulas to NBW. We start with a translation that goes via ABWs. For completeness, we also present the original translation of [77], which directly generates NBWs. The translation involves an exponential blow-up, which we show to be tight.

4.6.1.1 A Translation via ABWs

Consider an LTL formula ψ. For simplicity, we assume that ψ is given in *positive normal form*. Thus, negation is applied only to atomic propositions. Formally, given a set AP of atomic propositions, an LTL formula in positive normal form is defined as follows:

- $\mathbf{true}, \mathbf{false}$, p, or $\neg p$, for $p \in AP$.
- ψ_1, $\psi_1 \wedge \psi_2$, $\psi_1 \vee \psi_2$, $X\psi_1$, $\psi_1 U\psi_2$, or $\psi_1 R\psi_2$, where ψ_1 and ψ_2 are LTL formulas in positive normal form.

Note that the fact negation is restricted to atomic propositions has forced us to add not only the Boolean operator $\vee$ but also the temporal operator R. Still, it is easy to see that transforming an LTL formula to a formula in positive normal form involves no blow-up. The *closure* of an LTL formula ψ, denoted $cl(\psi)$, is the set of all its subformulas. Formally, $cl(\psi)$ is the smallest set of formulas that satisfy the following.

- $\psi \in cl(\psi)$.

	$\{p\}$	$\emptyset$
GFp	GFp	$GFp \wedge Fp$
Fp	***true***	Fp

Fig. 8 The transition function of an ABW for GFp

- If $\psi_1 \wedge \psi_2$, $\psi_1 \vee \psi_2$, $\psi_1 U \psi_2$ or $\psi_1 R \psi_2$ is in $cl(\psi)$, then $\psi_1 \in cl(\psi)$ and $\psi_2 \in cl(\psi)$.
- If $X\psi_1$ is in $cl(\psi)$, then $\psi_1 \in cl(\psi)$.

For example, $cl(p \wedge ((Xp)Uq))$ is $\{p \wedge ((Xp)Uq), p, (Xp)Uq, Xp, q\}$. It is easy to see that the size of $cl(\psi)$ is linear in $|\psi|$.

Theorem 24 *For every LTL formula ψ, there is an ABW $\mathscr{A}_\psi$ with $O(|\psi|)$ states such that $\mathscr{L}(\mathscr{A}_\psi) = \mathscr{L}(\psi)$.*

Proof We define $\mathscr{A}_\psi = \langle 2^{AP}, cl(\psi), \psi, \delta, \alpha \rangle$, where

- The transition $\delta(\varphi, \sigma)$ is defined according to the form of φ as follows.
  - $\delta(p, \sigma) = \begin{cases} \textbf{\textit{true}} \text{ if } p \in \sigma, \\ \textbf{\textit{false}} \text{ if } p \notin \sigma. \end{cases}$
  - $\delta(\neg p, \sigma) = \begin{cases} \textbf{\textit{true}} \text{ if } p \notin \sigma, \\ \textbf{\textit{false}} \text{ if } p \in \sigma. \end{cases}$
  - $\delta(\varphi_1 \wedge \varphi_2, \sigma) = \delta(\varphi_1, \sigma) \wedge \delta(\varphi_2, \sigma)$.
  - $\delta(\varphi_1 \vee \varphi_2, \sigma) = \delta(\varphi_1, \sigma) \vee \delta(\varphi_2, \sigma)$.
  - $\delta(X\varphi, \sigma) = \varphi$.
  - $\delta(\varphi_1 U \varphi_2, \sigma) = \delta(\varphi_2, \sigma) \vee (\delta(\varphi_1, \sigma) \wedge \varphi_1 U \varphi_2)$.
  - $\delta(\varphi_1 R \varphi_2, \sigma) = \delta(\varphi_2, \sigma) \wedge (\delta(\varphi_1, \sigma) \vee \varphi_1 R \varphi_2)$.
- The set α of accepting states consists of all the formulas in $cl(\psi)$ of the form $\varphi_1 R \varphi_2$.

The proof of the correctness of the construction proceeds by induction on the structure of ψ. For a formula $\varphi \in cl(\psi)$, we prove that when $\mathscr{A}_\psi$ is in state φ, it accepts exactly all words that satisfy φ. The base case, when φ is an atomic proposition or its negation, follows from the definition of the transition function. The other cases follow from the semantics of LTL and the induction hypothesis. In particular, the definition of α guarantees that in order for a word to satisfy $\varphi_1 U \varphi_2$, it must have a suffix that satisfies φ_2. Indeed, otherwise, the run of $\mathscr{A}_\psi$ has an infinite path that remains forever in the state $\varphi_1 U \varphi_2$, and thus does not satisfy α. □

Example 5 We describe an ABW $\mathscr{A}_\psi$ for the LTL formula $\psi = GFp$. Note that $\psi = \textbf{\textit{false}} R (\textbf{\textit{true}} U p)$. In the example, we use the F and G abbreviations. The alphabet of $\mathscr{A}_\psi$ consists of the two letters in $2^{\{p\}}$. The set of accepting states is $\{GFp\}$, and the states and transitions are described in the table in Fig. 8.

Example 6 We describe an ABW $\mathscr{A}_\psi$ for the LTL formula $\psi = p \wedge ((Xp)Uq)$. The alphabet of $\mathscr{A}_\psi$ consists of the four letters in $2^{\{p,q\}}$. The states and transitions are

	$\{p,q\}$	$\{p\}$	$\{q\}$	$\emptyset$
$p \wedge ((Xp)Uq))$	***true***	$p \wedge ((Xp)Uq)$	***false***	***false***
p	***true***	***true***	***false***	***false***
$(Xp)Uq$	***true***	$p \wedge ((Xp)Uq))$	***true***	$p \wedge ((Xp)Uq))$

Fig. 9 The transition function of an ABW for $p \wedge ((Xp)Uq))$

described in the table in Fig. 9. No state is accepting. Note that only the initial state is reachable.

Combining Theorems 24 and 22, we get the following.

Theorem 25 *For every LTL formula ψ, there is an NBW $\mathcal{A}_\psi$ such that $\mathcal{L}(\mathcal{A}_\psi) = \mathcal{L}(\psi)$ and the size of $\mathcal{A}_\psi$ is exponential in $|\psi|$.*

In Sect. 4.6.1.3 we show a matching exponential lower bound. Let us note here that while the 3^n blow-up in Theorem 22 refers to general ABWs, the ABWs obtained from LTL in the proof of Theorem 24 have a special structure: all the cycles in the automata are self-loops. For such automata (termed *very-weak* alternating automata, as they are weak alternating automata in which all SCCs are singletons), alternation can be removed with only an $n2^n$ blow-up [3, 20].

4.6.1.2 A Direct Translation to NBWs

The original translation of LTL to NBW [77] does not go via intermediate alternating automata. For completeness, we detail it here. The translation does not assume a positive normal form, and uses the *extended closure* of the given formula: For an LTL formula ψ, the extended closure of ψ, denoted $ecl(\psi)$, is the set of ψ's subformulas and their negations ($\neg\neg\psi$ is identified with ψ). Formally, $ecl(\psi)$ is the smallest set of formulas that satisfy the following.

- $\psi \in ecl(\psi)$.
- If $\psi_1 \in ecl(\psi)$ then $\neg\psi_1 \in ecl(\psi)$.
- If $\neg\psi_1 \in ecl(\psi)$ then $\psi_1 \in ecl(\psi)$.
- If $\psi_1 \wedge \psi_2 \in ecl(\psi)$ then $\psi_1 \in ecl(\psi)$ and $\psi_2 \in ecl(\psi)$.
- If $X\psi_1 \in ecl(\psi)$ then $\psi_1 \in ecl(\psi)$.
- If $\psi_1 U \psi_2 \in ecl(\psi)$ then $\psi_1 \in ecl(\psi)$ and $\psi_2 \in ecl(\psi)$.

For example, $ecl(p \wedge ((Xp)Uq))$ is $\{p \wedge ((Xp)Uq), \neg(p \wedge ((Xp)Uq)), p, \neg p, (Xp)Uq, \neg((Xp)Uq), Xp, \neg Xp, q, \neg q\}$.

The translation is based on the observation that the question of satisfaction of an LTL formula ψ in a computation π can be reduced to questions about the satisfaction of formulas in $ecl(\psi)$ in the suffixes of π. More formally, given a computation π, it is possible to (uniquely) label each suffix of π by the subset of formulas in $ecl(\psi)$ that are satisfied in this suffix. The correctness of this labeling can be verified by local consistency checks, which relate the labeling of successive suffixes, and by

a global consistency check, which takes care of satisfaction of eventualities. Since it is easier to check the satisfaction of each eventuality in isolation, we describe a translation to nondeterministic automata with the generalized Büchi acceptance condition. One can then use Theorem 12 in order to obtain an NBW.

Formally, given an LTL formula ψ over AP, we define $\mathscr{A}_\psi = \langle 2^{AP}, Q, Q_0, \delta, \alpha \rangle$, as follows.

- We say that a set $S \subseteq ecl(\psi)$ is *good in* $ecl(\psi)$ if S is a maximal set of formulas in $ecl(\psi)$ that does not have propositional inconsistency. Thus, S satisfies the following conditions.
  1. For all $\psi_1 \in ecl(\psi)$, we have $\psi_1 \in S$ iff $\neg\psi_1 \notin S$, and
  2. For all $\psi_1 \wedge \psi_2 \in ecl(\psi)$, we have $\psi_1 \wedge \psi_2 \in S$ iff $\psi_1 \in S$ and $\psi_2 \in S$.

  The state space $Q \subseteq 2^{ecl(\psi)}$ is the set of all the good sets in $ecl(\psi)$.
- Let S and S' be two good sets in $ecl(\psi)$, and let $\sigma \subseteq AP$ be a letter. Then $S' \in \delta(S, \sigma)$ if the following hold.
  1. $\sigma = S \cap AP$,
  2. For all $X\psi_1 \in ecl(\psi)$, we have $X\psi_1 \in S$ iff $\psi_1 \in S'$, and
  3. For all $\psi_1 U \psi_2 \in ecl(\psi)$, we have $\psi_1 U \psi_2 \in S$ iff either $\psi_2 \in S$ or both $\psi_1 \in S$ and $\psi_1 U \psi_2 \in S'$.

  Note that the last condition also means that for all $\neg(\psi_1 U \psi_2) \in ecl(\psi)$, we have that $\neg(\psi_1 U \psi_2) \in S$ iff $\neg\psi_2 \in S$ and either $\neg\psi_1 \in S$ or $\neg(\psi_1 U \psi_2) \in S'$.
- $Q_0 \subseteq Q$ is the set of all states $S \in Q$ for which $\psi \in S$.
- Every formula $\psi_1 U \psi_2$ contributes to the generalized Büchi condition α the set

$$\alpha_{\psi_1 U \psi_2} = \{S \in Q : \psi_2 \in S \text{ or } \neg(\psi_1 U \psi_2) \in S\}.$$

We now turn to discuss the size of $\mathscr{A}_\psi$. It is easy to see that the size of $ecl(\psi)$ is $O(|\psi|)$, so $\mathscr{A}_\psi$ has $2^{O(|\psi|)}$ states. Note that since ψ has at most $|\psi|$ subformulas of the form $\psi_1 U \psi_2$, the index of α is at most $|\psi|$. It follows from Theorem 12 that ψ can also be translated into an NBW with $2^{O(|\psi|)}$ states.

Remark 4 Note that the construction of $\mathscr{A}_\psi$ can proceed *on-the-fly*. Thus, given a state S of $\mathscr{A}_\psi$ and a letter $\sigma \in 2^{AP}$, it is possible to compute the set $\delta(S, \sigma)$ based on the formulas in S. As we shall see in Sect. 4.6.2, this fact is very helpful, as it implies that reasoning about $\mathscr{A}_\psi$ need not construct the whole state space of $\mathscr{A}_\psi$ but can rather proceed in an on-demand fashion.

4.6.1.3 The Blow-up in the LTL to NBW Translation

In this section we describe an exponential lower bound for the translation of LTL to NBW, implying that the blow-up that both translations above involve cannot in general be avoided. We do so by describing a doubly-exponential lower bound for the translation of LTL to DBW. Recall that NBWs are strictly more expressive than DBWs. The expressiveness gap carries over to languages that can be specified in

LTL. For example, the formula FGb ("eventually always b", which is similar to the language used in the proof of Theorem 6), cannot be translated into a DBW. We now show that when a translation exists, it is doubly-exponential. Thus, the exponential blow-ups in Theorem 25 and determinization (when possible) are additive:

Theorem 26 *When possible, the translation of LTL formulas to deterministic Büchi automata is doubly-exponential.*

Proof Let ψ be an LTL formula of length n and let $\mathcal{A}_\psi$ be an NBW that recognizes ψ. By Theorem 25, the automaton $\mathcal{A}_\psi$ has $2^{O(n)}$ states. By determinizing $\mathcal{A}_\psi$, we get a DPW $\mathcal{U}_\psi$ with $2^{2^{O(n)}}$ states [59, 64]. By Büchi typeness of DPWs [35] (see also Theorem 9), if $\mathcal{U}_\psi$ has an equivalent DBW, it can be translated into a DBW with the same state space. Hence the upper bound.

For the lower bound, consider the following ω-regular language L_n over the alphabet $\{0, 1, \#, \$\}$:[4]

$$L_n = \left\{ \{0,1,\#\}^* \cdot \# \cdot w \cdot \# \cdot \{0,1,\#\}^* \cdot \$ \cdot w \cdot \#^\omega \ : w \in \{0,1\}^n \right\}.$$

A word τ is in L_n iff the suffix of length n that comes after the single \$ in τ appears somewhere before the \$. By [7], the smallest deterministic automaton that accepts L_n has at least 2^{2^n} states. (The proof in [7] considers the language of the finite words obtained from L_n by omitting the $\#^\omega$ suffix. The proof, however, is independent of this technical detail: reaching the \$, the automaton should remember the possible set of words in $\{0,1\}^n$ that have appeared before.) We can specify L_n with an LTL formula of length quadratic in n. The formula is a conjunction of two formulas. The first conjunct, ψ_1, makes sure that there is only one \$ in the word, followed by a word in $\{0,1\}^n$, which is followed by an infinite tail of #'s. The second conjunct, ψ_2, states that eventually there exists a position with # and for all $1 \leq i \leq n$, the i-th letter from this position is 0 or 1 and it agrees with the i-th letter after the \$. Also, the $(n+1)$-th letter from this position is #. Formally,

- $\psi_1 = (\neg\$)U(\$ \wedge X((0 \vee 1) \wedge X(0 \vee 1) \wedge \overset{n}{\cdots} X((0 \vee 1) \wedge XG\#))) \cdots)$.
- $\psi_2 = F(\# \wedge X^{n+1}\# \wedge \bigwedge_{1 \leq i \leq n}((X^i 0 \wedge G(\$ \rightarrow X^i 0)) \vee (X^i 1 \wedge G(\$ \rightarrow X^i 1))))$.

Note that the argument about the size of the smallest deterministic automaton that recognizes L_n is independent of the automaton's acceptance condition. Thus, the theorem holds for deterministic Rabin, Streett, and Muller automata as well. □

4.6.2 Model Checking and Satisfiability

In this section we describe the automata-theoretic approach to LTL satisfiability and model checking. We show how, using the translation of LTL into NBW, these problems can be reduced to problems about automata and their languages.

[4] Note that, for technical convenience, the alphabet of L_n is not of the form 2^{AP}. It is easy to adjust the proof to this setting, say by encoding $\{0, 1, \#, \$\}$ by two atomic propositions.

Theorem 27 *The LTL satisfiability problem is PSPACE-complete.*

Proof An LTL formula ψ is satisfiable iff the automaton $\mathcal{A}_\psi$ is not empty. Indeed, $\mathcal{A}_\psi$ accepts exactly all the computations that satisfy ψ. By Theorem 16, the non-emptiness problem for NBWs is in NLOGSPACE. Since the size of $\mathcal{A}_\psi$ is exponential in $|\psi|$, and its construction can be done on-the-fly, membership in PSPACE follows. Hardness in PSPACE is proved in [69], and the proof is similar to the hardness proof we detailed for NBW non-universality in Theorem 17. Indeed, as there, given a polynomial-space Turing machine T, we can construct an LTL formula ψ_T of polynomial size that describes exactly all words that either do not encode a legal computation of T on the empty tape, or encode a rejecting computation. The formula $\neg\psi$ is then satisfiable iff T accepts the empty tape. □

Theorem 28 *The LTL model-checking problem is PSPACE-complete.*

Proof Consider a Kripke structure $K = \langle AP, W, W_0, R, \ell\rangle$. We construct an NBW $\mathcal{A}_K$ such that $\mathcal{A}_K$ accepts a computation $\pi \in (2^{AP})^\omega$ iff π is a computation of K. The construction of $\mathcal{A}_K$ essentially moves the labels of K from the states to the transitions. Thus, $\mathcal{A}_K = \langle 2^{AP}, W, W_0, \delta, W\rangle$, where for all $w \in W$ and $\sigma \in 2^{AP}$, we have

$$\delta(w,\sigma) = \begin{cases} \{w' : R(w,w')\} & \text{if } \sigma = \ell(w), \\ \emptyset & \text{if } \sigma \neq \ell(w). \end{cases}$$

Now, K satisfies ψ iff all the computations of K satisfy ψ, thus $\mathcal{L}(\mathcal{A}_K) \subseteq \mathcal{L}(\mathcal{A}_\psi)$. A naive check of the above would construct $\mathcal{A}_\psi$ and complement it. Complementation, however, involves an exponential blow-up, on top of the exponential blow-up in the translation of ψ to $\mathcal{A}_\psi$. Instead, we exploit the fact that LTL formulas are easy to complement and check that $\mathcal{L}(\mathcal{A}_K) \cap \mathcal{L}(\mathcal{A}_{\neg\psi}) = \emptyset$, where $\mathcal{A}_{\neg\psi}$ is the NBW for $\neg\psi$. Accordingly, the model-checking problem can be reduced to the non-emptiness problem of the intersection of A_K and $\mathcal{A}_{\neg\psi}$. Let $\mathcal{A}_{K,\neg\psi}$ be an NBW accepting the intersection of the languages of $\mathcal{A}_K$ and $\mathcal{A}_{\neg\psi}$. Since $\mathcal{A}_K$ has no acceptance condition, the construction of $\mathcal{A}_{K,\neg\psi}$ can proceed by simply taking the product of $\mathcal{A}_K$ with $\mathcal{A}_{\neg\psi}$. Then, K satisfies ψ iff $\mathcal{A}_{K,\neg\psi}$ is empty. By Theorem 25, the size of $\mathcal{A}_{\neg\psi}$ is exponential in $|\psi|$. Also, the size of $\mathcal{A}_K$ is linear in $|K|$. Thus, the size of $\mathcal{A}_{K,\neg\psi}$ is $|K| \cdot 2^{O(|\psi|)}$. Since the construction of $\mathcal{A}_{\neg\psi}$, and hence also $\mathcal{A}_{K,\neg\psi}$, can be done on-the-fly, membership in PSPACE follows from the membership in NLOGSPACE of the non-emptiness problem for NBW. Hardness in PSPACE is proved in [69], and again, proceeds by a generic reduction from polynomial-space Turing machines. □

As described in the proof of Theorem 28, the PSPACE complexity of the LTL model-checking problem follows from the exponential size of the product automaton $\mathcal{A}_{K,\neg\psi}$. Note that $\mathcal{A}_{K,\neg\psi}$ is exponential only in $|\psi|$, and is linear in $|K|$. Nevertheless, as K is typically much bigger than ψ, and the exponential blow-up of the translation of ψ to $\mathcal{A}_{\neg\psi}$ rarely appears in practice, it is the linear dependency in $|K|$, rather than the exponential dependency in $|\psi|$, that makes LTL model checking so challenging in practice.

We note that the translations described in Sect. 4.6.1 are the classic ones. Since their introduction, researchers have suggested many heuristics and optimizations, with rapidly changing state of the art. Prominent ideas involve a reduction of the state space by associating states with smaller subsets of the closure [21], possibly as a result of starting with alternating automata [20, 46], reductions based on relations between the states, in either the alternating or nondeterministic automaton [19, 71], working with acceptance conditions that are defined with respect to edges rather than states [22], and a study of easy fragments [34]. In addition, variants of NBWs are used for particular applications, such as *testers* in the context of composition reasoning [60]. Finally, the automata-theoretic approach has been extended also to *branching temporal logics*. There, formulas are interpreted over branching structures, and the techniques are based on automata on infinite trees [12, 13, 46, 57, 76].

Acknowledgement I thank Javier Esparza and Moshe Y. Vardi for many helpful comments and discussions.

References

1. Alur, R., Henzinger, T.A., Kupferman, O.: Alternating-time temporal logic. In: Proc. 38th IEEE Symp. on Foundations of Computer Science, pp. 100–109 (1997)
2. Armoni, R., Fix, L., Flaisher, A., Gerth, R., Ginsburg, B., Kanza, T., Landver, A., Mador-Haim, S., Singerman, E., Tiemeyer, A., Vardi, M.Y., Zbar, Y.: The ForSpec temporal logic: a new temporal property-specification logic. In: Proc. 8th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 2280, pp. 196–211. Springer, Heidelberg (2002)
3. Boker, U., Kupferman, O., Rosenberg, A.: Alternation removal in Büchi automata. In: Proc. 37th Int. Colloq. on Automata, Languages, and Programming, vol. 6199, pp. 76–87 (2010)
4. Boker, U., Kupferman, O., Steinitz, A.: Parityzing Rabin and Streett. In: Proc. 30th Conf. on Foundations of Software Technology and Theoretical Computer Science, pp. 412–423 (2010)
5. Breuers, S., Löding, C., Olschewski, J.: Improved Ramsey-based Büchi complementation. In: Proc. 15th Int. Conf. on Foundations of Software Science and Computation Structures. LNCS, vol. 7213, pp. 150–164. Springer, Heidelberg (2012)
6. Büchi, J.R.: On a decision method in restricted second order arithmetic. In: Proc. Int. Congress on Logic, Method, and Philosophy of Science. 1960, pp. 1–12. Stanford University Press, Stanford (1962)
7. Chandra, A.K., Kozen, D.C., Stockmeyer, L.J.: Alternation. J. Assoc. Comput. Mach. **28**(1), 114–133 (1981)
8. Choueka, Y.: Theories of automata on ω-tapes: a simplified approach. J. Comput. Syst. Sci. **8**, 117–141 (1974)
9. Cimatti, A., Clarke, E.M., Giunchiglia, F., Roveri, M.: NuSMV: a new symbolic model checker. Int. J. Softw. Tools Technol. Transf. **2**(4), 410–425 (2000)
10. Cormen, T.H., Leiserson, C.E., Rivest, R.L.: Introduction to Algorithms. MIT Press/McGraw-Hill, Cambridge/New York (1990)
11. Couvreur, J.-M.: On-the-fly verification of linear temporal logic. In: World Congress on Formal Methods. LNCS, vol. 1708, pp. 253–271. Springer, Heidelberg (1999)
12. Emerson, E.A., Jutla, C.: The complexity of tree automata and logics of programs. In: Proc. 29th IEEE Symp. on Foundations of Computer Science, pp. 328–337 (1988)
13. Emerson, E.A., Jutla, C.: Tree automata, μ-calculus and determinacy. In: Proc. 32nd IEEE Symp. on Foundations of Computer Science, pp. 368–377 (1991)

14. Emerson, E.A., Lei, C.-L.: Modalities for model checking: branching time logic strikes back. In: Proc. 12th ACM Symp. on Principles of Programming Languages, pp. 84–96 (1985)
15. Emerson, E.A., Lei, C.-L.: Temporal model checking under generalized fairness constraints. In: Proc. 18th Hawaii Int. Conf. on System Sciences. Western Periodicals Company, North Hollywood (1985)
16. Emerson, E.A., Lei, C.-L.: Modalities for model checking: branching time logic strikes back. Sci. Comput. Program. **8**, 275–306 (1987)
17. Fogarty, S., Kupferman, O., Vardi, M.Y., Wilke, T.: Unifying Büchi complementation constructions. In: Proc. 20th Annual Conf. of the European Association for Computer Science Logic, pp. 248–263 (2011)
18. Friedgut, E., Kupferman, O., Vardi, M.Y.: Büchi complementation made tighter. Int. J. Found. Comput. Sci. **17**(4), 851–868 (2006)
19. Fritz, C.: Constructing Büchi automata from linear temporal logic using simulation relations for alternating Büchi automata. In: Proc. 8th Int. Conf. on Implementation and Application of Automata. LNCS, vol. 2759, pp. 35–48. Springer, Heidelberg (2003)
20. Gastin, P., Oddoux, D.: Fast LTL to Büchi automata translation. In: Proc. 13th Int. Conf. on Computer Aided Verification. LNCS, vol. 2102, pp. 53–65. Springer, Heidelberg (2001)
21. Gerth, R., Peled, D., Vardi, M.Y., Wolper, P.: Simple on-the-fly automatic verification of linear temporal logic. In: Dembiski, P., Sredniawa, M. (eds.) Protocol Specification, Testing, and Verification, pp. 3–18. Chapman & Hall, London (1995)
22. Giannakopoulou, D., Lerda, F.: From states to transitions: improving translation of LTL formulae to Büchi automata. In: Proc. 22nd International Conference on Formal Techniques for Networked and Distributed Systems. LNCS, vol. 2529, pp. 308–326. Springer, Heidelberg (2002)
23. Godefroid, P., Wolper, P.: A partial approach to model checking. Inf. Comput. **110**(2), 305–326 (1994)
24. Hardin, R.H., Har'el, Z., Kurshan, R.P.: COSPAN. In: Proc. 8th Int. Conf. on Computer Aided Verification. LNCS, vol. 1102, pp. 423–427. Springer, Heidelberg (1996)
25. Henzinger, M., Telle, J.A.: Faster algorithms for the nonemptiness of Streett automata and for communication protocol pruning. In: Proc. 5th Scandinavian Workshop on Algorithm Theory. LNCS, vol. 1097, pp. 10–20. Springer, Heidelberg (1996)
26. Henzinger, T.A., Kupferman, O., Vardi, M.Y.: A space-efficient on-the-fly algorithm for real-time model checking. In: Proc. 7th Int. Conf. on Concurrency Theory. LNCS, vol. 1119, pp. 514–529. Springer, Heidelberg (1996)
27. Holzmann, G.J.: The model checker SPIN. IEEE Trans. Softw. Eng. **23**(5), 279–295 (1997)
28. Immerman, N.: Nondeterministic space is closed under complement. Inf. Comput. **17**, 935–938 (1988)
29. Jurdzinski, M.: Small progress measures for solving parity games. In: Proc. 17th Symp. on Theoretical Aspects of Computer Science. LNCS, vol. 1770, pp. 290–301. Springer, Heidelberg (2000)
30. Kähler, D., Wilke, T.: Complementation, disambiguation, and determinization of Büchi automata unified. In: Proc. 35th Int. Colloq. on Automata, Languages, and Programming. LNCS, vol. 5126, pp. 724–735. Springer, Heidelberg (2008)
31. Katz, S., Peled, D.: Interleaving set temporal logic. Theor. Comput. Sci. **75**(3), 263–287 (1990)
32. King, V., Kupferman, O., Vardi, M.Y.: On the complexity of parity word automata. In: Proc. 4th Int. Conf. on Foundations of Software Science and Computation Structures. LNCS, vol. 2030, pp. 276–286. Springer, Heidelberg (2001)
33. Klarlund, N.: Progress measures for complementation of ω-automata with applications to temporal logic. In: Proc. 32nd IEEE Symp. on Foundations of Computer Science, pp. 358–367 (1991)
34. Kretínský, J., Esparza, J.: Deterministic automata for the (F, G)-fragment of LTL. In: Proc. 24th Int. Conf. on Computer Aided Verification. LNCS, vol. 7358, pp. 7–22. Springer, Heidelberg (2012)

35. Krishnan, S.C., Puri, A., Brayton, R.K.: Deterministic ω-automata vis-a-vis deterministic Büchi automata. In: Algorithms and Computations. LNCS, vol. 834, pp. 378–386. Springer, Heidelberg (1994)
36. Kupferman, O.: Avoiding determinization. In: Proc. 21st IEEE Symp. on Logic in Computer Science, pp. 243–254 (2006)
37. Kupferman, O.: Sanity checks in formal verification. In: Proc. 17th Int. Conf. on Concurrency Theory. LNCS, vol. 4137, pp. 37–51. Springer, Heidelberg (2006)
38. Kupferman, O., Morgenstern, G., Murano, A.: Typeness for ω-regular automata. Int. J. Found. Comput. Sci. **17**(4), 869–884 (2006)
39. Kupferman, O., Piterman, N., Vardi, M.Y.: Extended temporal logic revisited. In: Proc. 12th Int. Conf. on Concurrency Theory. LNCS, vol. 2154, pp. 519–535 (2001)
40. Kupferman, O., Piterman, N., Vardi, M.Y.: Model checking linear properties of prefix-recognizable systems. In: Proc. 14th Int. Conf. on Computer Aided Verification. LNCS, vol. 2404, pp. 371–385. Springer, Heidelberg (2002)
41. Kupferman, O., Vardi, M.Y.: On the complexity of branching modular model checking. In: Proc. 6th Int. Conf. on Concurrency Theory. LNCS, vol. 962, pp. 408–422. Springer, Heidelberg (1995)
42. Kupferman, O., Vardi, M.Y.: Verification of fair transition systems. In: Proc. 8th Int. Conf. on Computer Aided Verification. LNCS, vol. 1102, pp. 372–382. Springer, Heidelberg (1996)
43. Kupferman, O., Vardi, M.Y.: An automata-theoretic approach to reasoning about infinite-state systems. In: Proc. 12th Int. Conf. on Computer Aided Verification. LNCS, vol. 1855, pp. 36–52. Springer, Heidelberg (2000)
44. Kupferman, O., Vardi, M.Y.: Weak alternating automata are not that weak. ACM Trans. Comput. Log. **2**(2), 408–429 (2001)
45. Kupferman, O., Vardi, M.Y.: Safraless decision procedures. In: Proc. 46th IEEE Symp. on Foundations of Computer Science, pp. 531–540 (2005)
46. Kupferman, O., Vardi, M.Y., Wolper, P.: An automata-theoretic approach to branching-time model checking. J. ACM **47**(2), 312–360 (2000)
47. Kurshan, R.P.: Complementing deterministic Büchi automata in polynomial time. J. Comput. Syst. Sci. **35**, 59–71 (1987)
48. Kurshan, R.P.: Computer Aided Verification of Coordinating Processes. Princeton University Press, Princeton (1994)
49. Landweber, L.H.: Decision problems for ω-automata. Math. Syst. Theory **3**, 376–384 (1969)
50. Löding, C.: Optimal bounds for the transformation of omega-automata. In: Proc. 19th Conf. on Foundations of Software Technology and Theoretical Computer Science. LNCS, vol. 1738, pp. 97–109 (1999)
51. Maler, O., Staiger, L.: On syntactic congruences for ω-languages. Theor. Comput. Sci. **183**(1), 93–112 (1997)
52. McNaughton, R.: Testing and generating infinite sequences by a finite automaton. Inf. Control **9**, 521–530 (1966)
53. Meyer, A.R., Stockmeyer, L.J.: The equivalence problem for regular expressions with squaring requires exponential time. In: Proc. 13th IEEE Symp. on Switching and Automata Theory, pp. 125–129 (1972)
54. Michel, M.: Complementation is More Difficult with Automata on Infinite Words. CNET, Paris (1988)
55. Miyano, S., Hayashi, T.: Alternating finite automata on ω-words. Theor. Comput. Sci. **32**, 321–330 (1984)
56. Muller, D.E., Saoudi, A., Schupp, P.E.: Alternating automata, the weak monadic theory of the tree and its complexity. In: Proc. 13th Int. Colloq. on Automata, Languages, and Programming. LNCS, vol. 226, pp. 275–283. Springer, Heidelberg (1986)
57. Muller, D.E., Schupp, P.E.: Alternating automata on infinite trees. In: Automata on Infinite Words. LNCS, vol. 192, pp. 100–107. Springer, Heidelberg (1985)

58. Muller, D.E., Schupp, P.E.: Simulating alternating tree automata by nondeterministic automata: new results and new proofs of theorems of Rabin, McNaughton and Safra. Theor. Comput. Sci. **141**, 69–107 (1995)
59. Piterman, N.: From nondeterministic Büchi and Streett automata to deterministic parity automata. Log. Methods Comput. Sci. **3**(3), 5 (2007)
60. Pnueli, A., Zaks, A.: On the merits of temporal testers. In: 25 Years of Model Checking. LNCS, vol. 5000, pp. 172–195. Springer, Heidelberg (2008)
61. Rabin, M.O.: Decidability of second order theories and automata on infinite trees. Trans. Am. Math. Soc. **141**, 1–35 (1969)
62. Rabin, M.O.: Decidable theories. In: Barwise, J. (ed.) Handbook of Mathematical Logic, pp. 595–629. North-Holland, Amsterdam (1977)
63. Rabin, M.O., Scott, D.: Finite automata and their decision problems. IBM J. Res. Dev. **3**, 115–125 (1959)
64. Safra, S.: On the complexity of ω-automata. In: Proc. 29th IEEE Symp. on Foundations of Computer Science, pp. 319–327 (1988)
65. Safra, S., Vardi, M.Y.: On ω-automata and temporal logic. In: Proc. 21st ACM Symp. on Theory of Computing, pp. 127–137 (1989)
66. Schewe, S.: Büchi complementation made tight. In: Proc. 26th Symp. on Theoretical Aspects of Computer Science. LIPIcs, vol. 3, pp. 661–672. Schloss Dagstuhl—Leibniz-Zentrum fuer Informatik, Wadern (2009)
67. Schewe, S.: Tighter bounds for the determinisation of Büchi automata. In: Proc. 12th Int. Conf. on Foundations of Software Science and Computation Structures. LNCS, vol. 5504, pp. 167–181. Springer, Heidelberg (2009)
68. Schewe, S.: Beyond hyper-minimisation—minimising DBAs and DPAs is NP-complete. In: Proc. 30th Conf. on Foundations of Software Technology and Theoretical Computer Science. Leibniz International Proceedings in Informatics (LIPIcs), vol. 8, pp. 400–411 (2010)
69. Sistla, A.P., Clarke, E.M.: The complexity of propositional linear temporal logic. J. ACM **32**, 733–749 (1985)
70. Sistla, A.P., Vardi, M.Y., Wolper, P.: The complementation problem for Büchi automata with applications to temporal logic. Theor. Comput. Sci. **49**, 217–237 (1987)
71. Somenzi, F., Bloem, R.: Efficient Büchi automata from LTL formulae. In: Proc. 12th Int. Conf. on Computer Aided Verification. LNCS, vol. 1855, pp. 248–263. Springer, Heidelberg (2000)
72. Synopsys: Assertion-based verification (2003)
73. Tarjan, R.E.: Depth first search and linear graph algorithms. SIAM J. Comput. **1**(2), 146–160 (1972)
74. Thomas, W.: Automata on infinite objects. In: Handbook of Theoretical Computer Science, pp. 133–191 (1990)
75. Valmari, A.: A stubborn attack on state explosion. Form. Methods Syst. Des. **1**, 297–322 (1992)
76. Vardi, M.Y., Wolper, P.: Automata-theoretic techniques for modal logics of programs. J. Comput. Syst. Sci. **32**(2), 182–221 (1986)
77. Vardi, M.Y., Wolper, P.: Reasoning about infinite computations. Inf. Comput. **115**(1), 1–37 (1994)
78. Willems, B., Wolper, P.: Partial-order methods for model checking: from linear time to branching time. In: Proc. 11th IEEE Symp. on Logic in Computer Science, pp. 294–303 (1996)
79. Wolper, P., Vardi, M.Y., Sistla, A.P.: Reasoning about infinite computation paths. In: Proc. 24th IEEE Symp. on Foundations of Computer Science, pp. 185–194 (1983)
80. Yan, Q.: Lower bounds for complementation of ω-automata via the full automata technique. In: Proc. 33rd Int. Colloq. on Automata, Languages, and Programming. LNCS, vol. 4052, pp. 589–600. Springer, Heidelberg (2006)
81. Zielonka, W.: Infinite games on finitely coloured graphs with applications to automata on infinite trees. Theor. Comput. Sci. **200**(1–2), 135–183 (1998)

Chapter 5
Explicit-State Model Checking

Gerard J. Holzmann

Abstract In this chapter we discuss the methodology used in explicit-state logic model checking, specifically as applied to asynchronous software systems. As the name indicates, in an explicit-state model checker the state descriptor for a system is maintained in explicit, and not symbolic, form, as are all state transitions. Abstraction techniques and partial-order reduction algorithms are used to reduce the search space to a minimum, and advanced storage techniques can be used to extend the reach of this form of verification to very large system sizes. The basic algorithms for explicit-state model checking date from the late 1970s and early 1980s. More advanced versions of these algorithms remain an active area of research.

5.1 Introduction

There are many different approaches that can be used for the implementation of a model-checking procedure. One of the first methods, the origins of which we can trace back to basic reachability analysis techniques that were explored as early as in the 1970s, is explicit-state model checking. Explicit-state model checking turns out to be well suited for applications in software verification, specifically the verification of systems of interacting asynchronous processes. Explicit-state techniques have gained much of their power from their integration with partial-order reduction techniques (discussed in more detail in Chap. 6 of this Handbook [25]), which make it possible to limit the number of states that must be explored to prove or disprove a property by up to an exponential factor. This has made the use of explicit-state approaches with partial-order reduction for asynchronous software systems extremely competitive when compared with the use of symbolic model-checking techniques in the verification of, for instance, synchronous circuits [2, 7, 15].

A basic reachability analysis lends itself most easily to the verification of *safety* properties, such as the validity of invariants, assertions, or the absence of deadlock in a multi-process system. As we shall see though, explicit-state model-checking

G.J. Holzmann (✉)
Nimble Research, Monrovia, CA, USA
e-mail: gholzmann@acm.org

E.M. Clarke et al. (eds.), *Handbook of Model Checking*,
DOI 10.1007/978-3-319-10575-8_5

algorithms can also be used to prove *liveness* properties, including all properties that can be formalized in Linear Temporal Logic (LTL) and more broadly the class of all ω-regular properties.

An explicit-state model-checking procedure is only possible if a few important assumptions are satisfied. First, the system that is the target of the verification must be finite state. The state of the system is a, possibly abstracted, finite tuple of values, which can be chosen from any finite domain. The system can change *state* by executing *state transitions*. This means that we assume that the system can be represented as (a set of) finite state automata. A second assumption is that system execution can be modeled as a sequence of separate state transitions. This means that even when we describe systems with multiple processes, each modeled as a finite state automaton, the effect of a system execution can be modeled by an arbitrary *interleaving* of individual process actions. With this approach we can accurately represent process scheduling on a single CPU in a multi-threaded system, where the main CPU executes one instruction at a time, interleaving the actions of different processes based on scheduling decisions. It can, however, also represent process execution on multiple CPUs, e.g., in a multi-core system, or in a networked system. Note, for instance, that also in a multi-core system it is not possible for multiple processes to access a shared memory location truly simultaneously. Although the precise ordering of read and write operations cannot always be known, at some level of granularity it is always possible to determine an interleaving order that represents the actual execution sequence. The same is not necessarily true in asynchronous hardware circuits, which therefore require a different approach to model checking.

The first attempts to build automated tools for reachability analysis targeted simple finite-state descriptions of communication protocols. In 1979, Jan Hajek used a graph exploration tool [12] to verify properties of the protocols in Tanenbaum's primer on computer networks [27]. Around the same time, Colin West and Pitro Zafiropulo developed a reachability analysis procedure for the verification of another protocol, CCITT recommendation X.21, and identified a series of flaws [29, 30]. In 1980, the first version of what later became the logic model checker Spin was developed at Bell Labs. This tool, called *pan*, was successfully used in subsequent years to expose violations of safety properties in finite-state models of telephone switching systems and communication protocols [13].

5.1.1 The Importance of Abstraction

Clearly, if we model the executions of a software system in full detail, it will generally not be feasible to perform an exhaustive verification with an explicit-state method. As a very simple example to illustrate this, consider two asynchronously executing processes each containing a single 32-bit counter. If all the two processes do is to increment their counters, executing in a simple loop, then a fully detailed model would need to represent and explore $2^{(32+32)}$ or more than 10^{19} system states. Clearly, not all those 10^{19} states are relevant to specific correctness properties that we may be interested in proving about this system. We may, for instance, want to

prove that the system does or does not terminate, or that the counter-values can wrap around the maximum. For none of these properties will it be necessary to compute all possible combinations of values held by the two counters.

The early applications to protocol verification were successful because they could be focused on the *control*-aspects of a protocol and they could abstract from the *data*-aspects. By focusing on control, one can effectively prove correctness of data transmission across unreliable channels in a way that is independent of the actual data being transmitted. An elegant example of the use of this principle is known as Wolper's data independence theorem [31]. Similarly, if we model a mutual-exclusion algorithm, the detailed computations that may be performed both outside and inside the critical section that is to be protected from multiple access are irrelevant to the correctness of the algorithm itself, and can be abstracted during the verification.

The verification process begins with the identification of the *smallest sufficient model* that allows us to prove a property of a system. What the smallest sufficient model is will generally depend on the specific property to be proven. The smallest sufficient model is generally an abstraction of the *real* system, which we refer to as the *concrete system* in what follows. There are of course requirements on the type of abstractions that can be used in this step. It should, for instance, be impossible to prove a property about the abstract system that does not hold for the corresponding concrete system, that is: the abstraction should be logically sound. In addition to abstraction, a frequently used technique in the verification of complex systems is that of restriction or specialization. A restriction reduces the abstract system to a subset of behaviors in such a way that a counter-example to a correctness claim that is generated for the restricted system is also a counter-example of the non-restricted system, but the absence of a counter-example does not logically imply the absence of a counter-example also in the non-restricted system. In practice, both abstraction (generalization) and restriction (specialization) can play a critical role in managing the complexity of software verification with explicit-state model-checking techniques. We will return to this shortly, after we discuss the basic automata-theoretic framework and the main search algorithms that are used for explicit-state model checking. More on the use of abstraction techniques in applications of logic model checking can also be found in Chap. 13 of this Handbook [9].

5.2 Basic Search Algorithms

A basic reachability analysis algorithm, sufficient for proving safety properties, is readily implemented as either a breadth-first or depth-first search. The search can be done as a basic check on all system states that are reachable from a given initial state. System states then, can be thought of as the control-flow states and variable values of a program. When we apply a model-checking algorithm to a multi-threaded system, the system state is given as a combination of local process states, and the reachability graph is the interleaving product of process actions. We start by describing this interleaving product in a little more detail.

```
 1 Open D = {}; // typically an ordered set
 2 Visited V = {};
 3
 4 start()
 5 {    V!s0; D!s0;
 6      bfs();
 7 }
 8
 9 bfs()
10 {    while (D != {})
11      {    D?s;
12           check_validity(s);
13           foreach (s, e, s') in T
14           {    if !(s' ∈ V) { V!s'; D!s'; }
15 }    }    }
```

Fig. 1 Breadth-first search

Let $A = \{S, s_0, L, T, F\}$ be a *finite state automaton*, where S is a finite set of states, s_0 is an element of S called the initial state, L is a set of symbols called the *label* set or also the *alphabet*, $T \subseteq S \times L \times S$ is a set of transitions, and $F \subseteq S$ is the set of *final* states. Automaton A is said to *accept* any finite execution that ends in any state $s \in F$.

The *interleaving product* of finite state automata $A_0, \ldots, A_N$ is another finite state automaton $A' = \{S', s'_0, L', T', F'\}$, with

- $S' = A_0.S \times A_1.S \times \cdots \times A_N.S$,
- $s'_0 = (A_0.s_0, A_1.s_0, \ldots, A_N.s_0)$,
- $L' = A_0.L \cup A_1.L \cup \cdots \cup A_N.L$,
- $T' \subseteq S' \times L' \times S'$ such that for each transition $((A_0.s_0, A_1.s_1, \ldots, A_N.s_N), e, (A_0.t_0, A_1.t_1, \ldots, A_N.t_N)) \in T'$, we have $\exists i$, $0 \leq i \leq N$, $(A_i.s_i, e, A_i.t_i) \in A_i.T$, with $e \in A_i.L$ and $\forall j \neq i$, $A_j.s_j = A_j.t_j$, and
- $F' = A_0.F \times A_1.F \times \cdots \times A_N.F$.

A basic reachability algorithm can now be constructed as shown in Fig. 1, exploring and checking all states that are reachable from the initial state s_0 of a finite automaton as defined above. The algorithm uses two data structures, a set of *open* states D and a set of *visited* states V. Open states are states that are currently being explored, with not all of their successor states visited yet. We use the following notation for operations on sets:

- $X!y$ adds y to set X; if X is ordered then it adds y as the *last* element of X,
- $X!!y$ adds y to set X; if X is ordered then it adds y as the *first* element in X,
- $X?y$ removes an element from set X and names it y; if X is ordered then the element removed is the *first* element in X, if X is empty then the operation returns the null element Ø.

The search procedure illustrated in Fig. 1 generates all states reachable from initial state s_0, while checking for violations of safety properties by calling function `check_validity()` at each state, including s_0. Set V is an unordered set. For the correctness of the basic search procedure, it does not matter if the set of open states D is ordered or not. To get a breadth-first search discipline, though, D has to

```
 1 Open D = {}; // an ordered set
 2 Visited V = {};
 3
 4 start()
 5 {    V!s0; D!s0;
 6      dfs();
 7 }
 8
 9 dfs()
10 {    if (D != {})
11      {    D?s;
12           check_validity(s);
13           foreach (s, e, s') in T
14           {    if !(s' ∈ V) { V!s'; D!!s'; dfs(); }
15 }    }    }
```

Fig. 2 Depth-first search

be ordered and used as a queue: the retrieval operation on line 11 removes the first (and oldest) element, and the add operation on line 14 adds new elements at the end.

To make it possible to generate an example execution for any state that is found to violate the property (called a *counter-example*), we can extend set V to store a pointer back to the parent state whenever a new state is added. To generate the counter-example, we then only have to follow these back-pointers to find a path from the initial state to the violating state.

At the end of the breadth-first search, set D is empty, and set V contains all states reachable in a finite number of steps from initial state s_0.

The dual of breadth-first search is depth-first search. Depth-first search is most naturally written as a recursive procedure. It is illustrated in Fig. 2. There are two key changes in this version of the algorithm compared to breadth-first search. The first change is the recursive call that is placed in the inner loop, on line 14, after a newly generated state is added to the open and visited sets, and the matching change of the surrounding while-loop into an if-statement, on line 10. The second change is the change from $D!s'$ to $D!!s'$, on line 14, which means that instead of using D as a *queue*, we now use it as a *stack*. Generating a counter-example when a safety violation is found is now simple and requires no addition to the information stored in stack D: the execution is given by the contents of stack D at the point in the search that the safety violation is detected.

Although the change looks minor, and both algorithms have the same computational complexity (both are linear in the number of reachable states) the two algorithms behave very differently. In favor of depth-first search is that the amount of information that must be stored in set D to enable counter-example generation is smaller. In favor of breadth-first search is that any counter-example generated tends to be smaller than it is for depth-first search. In fact, it is easy to show that the counter-examples generated with a breadth-first search are the shortest possible. In this case then, we have to make a tradeoff between minimizing memory use, and thus being able to handle larger problem sizes, or shortening counter-examples, and thus making it easier to understand errors found.

Note that both algorithms can work *on the fly*, which means that the reachability graph (the automaton describing the global state space as a product of the

smaller automata that formalize individual process behaviors) need not be known a priori, provided that the state transition relations are given. Both breadth-first and depth-first search can generate the product automaton on the fly, and both may be terminated as soon as a counter-example is discovered.

An important advantage of depth-first search is that it can fairly easily be extended to support not only the verification of *safety* but also *liveness* properties, without increasing the computational complexity of the search, which remains linear in the number of reachable states. Before we can discuss this extension, though, we have to discuss the extension from finite automata to ω-automata, and the fundamental relation between ω-automata and temporal logic. Both topics are covered in greater detail in other chapters of this Handbook, so a general description will suffice here.

5.3 Linear Temporal Logic

A formula in linear temporal logic (LTL) is formally defined as follows. Let f and g be arbitrary LTL formulas and let p by an arbitrary *state formula*: a Boolean expression that can be evaluated to yield a truth value for any given system state.

```
f     ::=   p | true | false | (f) | f binop f | unop f
unop  ::=   □ | ◊ | !
binop ::=   U | ∧ | ∨ | ⇒ | ⇔
```

In this grammar we have used three temporal operators: the box operator $\Box$ (which is pronounced *always*), the diamond operator $\Diamond$ (pronounced *eventually*), and the U operator *until*. The symbol $\Rightarrow$ stands for logical implication and $\Leftrightarrow$ for logical equivalence.

The semantics of temporal logic formulas is defined over infinite sequences [26]. For simplicity, we restrict ourselves here to execution sequences that start in initial state s_0.

Let σ be an infinite execution sequence, defined as the sequence of states $\sigma = s_0, s_1, \ldots, s_i, \ldots$.

$\Box$ f holds at state s_i if and only if
f holds at all states $s_j \in \sigma$ with j $\geq$ i,

$\Diamond$ f holds at state s_i if and only if
f holds for at least one state $s_j \in \sigma$ with j $\geq$ i,

f U g holds at state s_i if and only if
either g holds at s_i, or f holds at s_i and f U g holds at s_{i+1}
(informally: f is true at least until g becomes true).

5.4 Omega Automata

Any LTL formula can be mechanically converted into a Büchi automaton that accepts precisely those execution sequences for which the LTL formula is satisfied [26, 28]. The algorithm that the Spin model checker uses to convert LTL formula into Büchi automata is described in [10, 11].

The definition of a Büchi automaton $B = \{S, s_0, L, T, F\}$ is similar to the definition of a finite automaton that we presented earlier, but has a different definition of *acceptance*. Büchi automaton B accepts execution sequence σ if and only σ contains infinitely many states from set F. This means that acceptance for Büchi automata is defined for *infinite* sequences only.

A real system model may of course have both finite and infinite executions. To allow us to reason about both infinite and finite executions within the same theoretical framework we can consider any finite sequence to be a special case of an infinite sequence by extending it with an infinite repetition of its final state. This infinite repetition then corresponds to what is commonly called a *stuttering* step (a no-op) that is repeated ad infinitum, without leaving the final state. Clearly then, a finite sequence can only qualify for Büchi acceptance if the final state being stuttered is itself from set F, which matches the notion of finite acceptance from before.

Given a system A, formalized as a finite automaton, and a Büchi automaton B, formalizing all the executions of A that satisfy an LTL formula, the model-checking problem can now be phrased as the problem of finding an accepting run in the intersection of the languages accepted by the two automata. The intersection of A and B is obtained by computing the synchronous product $A \times B$, which is a Büchi automaton.

The *synchronous product* of finite state automata A and B is finite state automaton $P = \{S', s'_0, L', T', F'\}$, where $S' = A.S \times B.S$, $s'_0 = (A.s_0, B.s_0)$, $L' = A.L \times B.L$, and $T' \in S' \times L' \times S'$ such that for each transition $((A.s, B.t), (e, f), (A.s', B.t')) \subseteq T'$ we have $(A.s, e, A.s') \in A.T$, and $(B.t, f, B.t') \in B.T$.

The set of final states $F' \subseteq A.S \times B.S$ such that for each pair $(e, f) \in F'$ we have $e \in A.F \wedge f \in B.F$, i.e., each component state is a final state in its original automaton.

Since the synchronous product of two finite state automata is also finite state, an infinite execution of the product is necessarily cyclic.

Given a set of finite automata $A_0, \ldots, A_N$ and an LTL formula f, the model-checking problem can now be formalized as follows.

1. Convert LTL formula f into the corresponding Büchi automaton B.
2. Compute the interleaving product A of $A_0, \ldots, A_N$.
3. Compute the synchronous product P of $A \times B$.
4. Find accepting runs of automaton P, using the Büchi acceptance rule.

Any runs accepted by automaton P correspond to executions of A that satisfy formula f. If we are interested in finding the potential violations of f, all we have to do is to begin this procedure by replacing f with $\neg f$, using logical negation.

In practice, instead of performing steps 2, 3, and 4 one at a time, it is more efficient to perform them in a single step, again adopting an on-the-fly procedure for computing and checking P until a counter-example is found.

The model checker Spin deviates on one important point from the standard definition of the synchronous product of two Büchi automata given above. To determine the set of final states, Spin considers a state in the product $A \times B$ to be final if *at least one* of A or B is in a final state. This conforms to the standard definition in

the (standard) case where the use of final state labels (called *accept* state labels in Spin) is restricted to the property automaton B, and *all* states in the interleaving product A are considered to be final. Using the alternative definition allows Spin to use a slightly more general framework that allows also the definition of accept state labels within any of the component automata $A_0, \ldots, A_N$ (and thus A). We can then also use the model-checking procedure to find accepting runs within A itself, without requiring the definition of a separate property automaton B.

5.5 Nested Depth-First Search

The challenge is then to find infinite accepting runs in finite state automaton P. Because the accepting run must contain at least one final state, the search problem can be phrased as follows: does there exist at least one reachable final state in P that is also reachable from itself? This problem can be solved efficiently with a nested depth-first search algorithm.

The first part of this search serves to identify all final states in P that are reachable from the initial system state, and the second (nested) part of the search serves to identify those final states from this set that are also reachable from themselves. The nested search can be performed in such a way that the cost in runtime increases only by a factor of maximally two. In the worst case, each reachable state is now visited twice, but they of course only need to be stored once, which means that the memory requirements of a nested depth-first search are no different than those of a standard depth-first search. The basic procedure is illustrated in Fig. 3. The algorithm was implemented in the Spin model checker in 1989 [8], and revised to support partial order reduction in 1995 [22].

The correctness of the algorithm follows from the following theorem [8].

Theorem 1 *If acceptance cycles exist, the nested depth-first search algorithm will report at least one such cycle.*

Proof Let r be the first accepting state encountered in the depth-first search that is also reachable from itself. The nested part of the search is initiated from this state after all its successor states have been explored.

First note that state r itself cannot be reachable from any other state that was previously entered into the second state space. Suppose there was such a state w. To be in the second state space w either is itself an accepting state, or it is reachable from an accepting state. Call that accepting state v. If r is reachable from w in the second state space then r is also reachable from v. But, if r is reachable from v in the second state space, it is also reachable from v in the first state space. There are now two cases to consider:

1. The path from v to r in the first state space does not contain states that appear on the depth-first search stack when v is first reached.

```
 1 Open D = {};  // ordered set
 2 Visited V = {};
 3
 4 State seed = nil;
 5
 6 start()
 7 {    V!s0,0; D!s0,0;
 8      ndfs();    // start the first search
 9 }
10
11 ndfs()
12 {    Bit b;     // b=0: first search, b=1: nested search
13      D?s,b;
14
15      foreach (s,e,s') in T
16      {    if (s' == seed)     // seed reachable from itself
17           {     liveness_violation(); return;
18           }
19           if !(s',b∈ V) // if s' not reached before
20           {    V!s',b; D!!s',b; ndfs(); // continue search
21      }    }
22
23      // in post-order, in first search only
24
25      if (s∈ F && b == 0)
26      {    seed = s;     // a reachable final state
27           D!!s,1;       // push s on stack D
28           ndfs();       // start the nested search
29           seed = nil;   // nested search completed
30 }    }
```

Fig. 3 Nested depth-first search

2. The path from v to r in the first state space does contain at least one state x that appears on the depth-first search stack when v is first reached.

In the first case, r would have been entered into the second state space before v, due to the post-order discipline, contradicting the assumption that v is entered before r. (Remember that both r and v are assumed to be accepting states.)

In the second case, v is necessarily an accepting state that is reachable from itself, which contradicts the assumption that r is the first such state entered into the second state space.

State r is reachable from all states on the path from r back to itself, and therefore none of those states can already be in the second state space when this search begins. The path therefore cannot be truncated and r is guaranteed to find itself in the successor tree that is explored in the nested part of the search. □

It should be noted that even though the nested depth-first search algorithm visits states up to two times, each state needs to be stored just once. The states can be labeled with two bits to indicate the state space where they were first encountered, so there is virtually no increase in memory use. Another advantage of this method is that the algorithm still works *on the fly*: errors are detected during the exploration of the state space, and the search process can be cut short as soon as the first error is found.

It is clear that the computational complexity of the nested depth-first search algorithm is strictly linear in the size of P (measured as the number of reachable states

in P). The size of P itself is at most equal to the Cartesian product of the state sets of A and B. The size of A, in turn, is at most equal to the product of the sizes of all component automata $A_0, \ldots, A_N$. Finally, in the worst case the size of B can be exponential in the number of temporal operators in formula f.

In most cases of practical interest, the size of B is not a major factor: it typically contains fewer than ten states. This is in part due to the fact that most LTL formula of practical interest have very few temporal operators: the meaning of longer formulas can be notoriously hard to determine. It is also due to the fact that LTL to Büchi automata conversion algorithms very rarely exhibit worst-case behavior, although it is certainly possible to find exceptions. The real bottleneck in explicit-state logic model checking is the potential size of A.

There are two main strategies to cope with this complexity. The first is the use of partial-order reduction theory in the computation of the interleaving product A. The use of this strategy can, in the best case, achieve an exponential reduction in the size of A. If implemented well, the strategy has no down side; even in the worst case where no reduction can be achieved it will then not introduce noticeable overhead. The partial-order reduction algorithm that was implemented in the Spin model checker is described in [21, 22]. Partial-order reduction strategies are discussed in Chap. 6 of this Handbook [25]. The second main strategy is the use of abstraction, which we will discuss later in this chapter. There is also a range of coding techniques that can be used for explicit-state verification to reduce the amount of information that is stored during the verification process, thus enabling the verification of very large problems. We will also discuss some of these strategies later in this chapter.

5.6 Abstraction

To support abstraction methods (cf. Chap. 13), we can make one final change to nested depth-first search. The revised algorithm is shown in Fig. 4. The abstraction function used in this context can define a symmetry reduction [5, 6, 23], or any other abstraction, provided that it preserves logical soundness (i.e., it does not allow us to prove anything that is not true, or to disprove something that is).

Let w, x, y, and z denote states, σ and τ denote sequences of states (paths), and σ_i be the i-th state in σ. Further, let $\rightarrow$ denote the transition relation, i.e., $x \rightarrow y$ means that there exists an e such that $(x, e, y) \in T$.

A symmetric relation $\sim$ on states is a *bisimulation* relation if it satisfies the following condition [24]:

$$\forall w, y, z : (w \sim y \wedge y \rightarrow z) \Rightarrow (\exists x : w \rightarrow x \wedge x \sim z) \tag{1}$$

This means that states w and y are bisimilar if, whenever there is a transition from y to z, there is also a successor x of w such that x and z are bisimilar. We say that paths σ and τ *correspond* when $\forall i : \sigma_i \sim \tau_i$.

```
Open D = {};  // ordered set
Visited V = {};

State seed = nil;

start()
{    V!f(s0),0; D!s0,0; // f is the abstraction function
     ndfs_abstract();   // start the first search
}

ndfs_abstract()
{     Bit b; // b=0: first search, b=1: nested search

     D?s,b; // s is a concrete (not abstract) state

     foreach (s,e,s') in T
     {    if (f(s') == seed) // seed reachable from itself
          {     liveness_violation(); return;
          }
          if !(f(s'),b∈ V) // f(s') not reached before
          {     V!f(s'),b; D!!s',b; ndfs(); // continue
     }    }

     // in post-order, in first search only

     if (s ∈ F && b == 0)
     {    seed = f(s);     // a reachable final state
          D!!s,1;          // push s on stack D
          ndfs_abstract(); // start the nested search
          seed = nil;      // nested search completed
}    }
```

Fig. 4 Nested depth-first search with abstraction function $f()$

Theorem 2 ([5]) *Let $\sim$ be a bisimulation relation, and let AP be a set of state formulas such that every $P \in AP$ satisfies the condition*

$$\forall x, y : (x \sim y) \Rightarrow \big(P(x) \Leftrightarrow P(y)\big) \tag{2}$$

then any two bisimilar states satisfy the same LTL formula over the propositions in AP.

This means that any abstraction that satisfies conditions (1) and (2) preserves the logical soundness of the LTL model-checking procedure [5]. We can make use of this by defining powerful abstractions in the verification of implementation-level code to reduce what otherwise would be an overwhelming amount of computational complexity.

5.6.1 Tic-Tac-Toe

We will describe an example of the use of this type of abstraction in explicit-state model checking as it is supported by the Spin model checker. Spin is a broadly used logic model-checking tool that is based on explicit-state techniques [15]. As an example we will use the familiar game of tic-tac-toe.

```
 1 mtype = { cross, circle };
 2
 3 typedef row { mtype c[3]; }
 4 typedef square { row r[3]; }
 5 square b;
 6
 7 #define match(x,y,z)  (b.r[x].c[y] == z)
 8 #define Row(y,z)      (match(0,y,z) && match(1,y,z) && match(2,y,z))
 9 #define Column(x,z)   (match(x,0,z) && match(x,1,z) && match(x,2,z))
10 #define Up(z)         (match(2,0,z) && match(1,1,z) && match(0,2,z))
11 #define Down(z)       (match(0,0,z) && match(1,1,z) && match(2,2,z))
12 #define horizontal(z) (Row(0,z) || Row(1,z) || Row(2,z))
13 #define vertical(z)   (Column(0,z) || Column(1,z) || Column(2,z))
14 #define diagonal(z)   (Up(z) || Down(z))
15 #define try(x,y,z)    b.r[x].c[y] == 0 -> b.r[x].c[y] = z
16
17 inline check_win(z) {
18      if
19      :: horizontal(z) || vertical(z) || diagonal(z) -> end_game: 0
20      :: else // continue the game
21      fi
22 }
23
24 inline place(z) {
25      if
26      :: try(0,0,z) :: try(0,1,z) :: try(0,2,z)
27      :: try(1,0,z) :: try(1,1,z) :: try(1,2,z)
28      :: try(2,0,z) :: try(2,1,z) :: try(2,2,z)
29      // if no moves are possible, it is a draw
30      fi
31 }
32
33 init {
34      mtype symbol = cross;
35 end: do
36      :: atomic {
37           place(symbol);
38           check_win(symbol);
39           symbol = (symbol == circle -> cross : circle);
40         }
41      od
42 }
```

Fig. 5 Pure Spin model for the game of tic-tac-toe

The game itself is easily modeled. We need to model the game board as a 3×3 square board. Each place can either be empty (represented by the initial value zero), or it can contain a cross or a circle. The game proceeds by selecting an arbitrary empty place on the board, placing a symbol (cross or circle) on that place, checking for a win, and then switching sides by alternating the symbol, as shown in Fig. 5. Spin starts by using the standard C preprocessor to interpret all macro definitions and include directives. In this example we have used only macro definitions. Both macros and inlines (two of the latter are used in Fig. 5) define a purely textual expansion of function names with optional parameter replacement. The inline $place(symbol)$, for instance, is an inline call where the parameter z from the inline definition is replaced with the text $symbol$.

The initial process (named $init$) contains an infinite loop ($do \ldots od$) that contains a single indivisibly executed sequence of statements. Inline $place$ defines a nondeterministic selection ($if \ldots fi$) of the nine possible moves that each player can

make. The inline *check* finally checks whether the game is won, and if it is it will bring the execution to a halt by attempting to execute the unexecutable statement 0.

Placing a symbol on a square is modeled as a non-deterministic selection of a free space on the board, and checking for a win is simply checking for a completed row, column, or diagonal involving the last-placed symbol.

Given three possible values for any one of the nine places on the game board we have maximally 3^9 = 19,683 possible board states. Not all of these states are reachable within the game. It is, for instance, not possible to fill all the places on the board with the same symbol.

The model checker explores 5,528 states for this version of the model. We have written the model in such a way that only draw or win states are counted for the total, by grouping all actions that are part of a single move inside an atomic statement. Clearly, though, there is still a significant amount of redundancy in this set.

For every unique state of the game board there are up to eight variations of what is essentially the same state that can be obtained by rotating or mirroring the board. We can define an abstraction function f that captures this equivalence relation on board states, to reduce the amount of work that the model checker has to do to just one member of each equivalence class. If we do so, it should be possible to reduce the number of states explored to a much smaller number, and to increase the efficiency of the verification process significantly.

We can encode the state of the game board as a nine-place ternary number, by assigning place values in a fixed order, e.g., but numbering the places on the board from one to nine starting at the top left, row by row, ending at the bottom right. This final number uniquely represents the board configuration. This board state is one of 16 equivalent states though, obtained by rotation and mirroring of the board. To obtain all 16 numbers all we have to do is to assign the place values in all 16 possible ways. The abstraction function can now be defined by simply choosing one canonical representative from each set of 16 board configurations. In our example we'll use the smallest of the 16 values for the abstraction. Complexity is not a significant concern in this small example, but we will also illustrate how the abstraction can be computed in a C function that can be integrated into the model.

The Spin model checker can define transitions either as statements in the specification language, or as standard blocks of C code. No special treatment is needed to support this capability since the C code fragments merely act as *state transformers*, just like any other type of statements. The extension that supports this is very powerful though, as we will illustrate.

The extended model for the tic-tac-toe example, using C code for the abstraction function, is shown in Fig. 6. The changes from Fig. 5 are shown in **bold** (lines 5–10, and 43). First, after each move we now call the abstraction function to compute the new abstract state. This is done with a call to a C function called `abstract_value()`, which is placed in an embedded C code statement. The function returns an integer value, the abstract representation of the game board, which we assign to a new global integer variable named `abstract` in the top-level model. To be able to refer to this variable from within a C code fragment, we have to prefix this variable with the name of the state-vector (called `now.`).

```
 1 mtype = { cross, circle };
 2
 3 typedef row { mtype c[3]; }
 4 typedef square { row r[3]; }
 5 hidden square b;
 6 c_track "&b" "sizeof(struct square)" "UnMatched";
 7 int abstract;
 8 c_code {
 9       \#include "abstraction.c"
10 }
11
12 #define match(x,y,z)  (b.r[x].c[y] == z)
13 #define Row(y,z)      (match(0,y,z) && match(1,y,z) && match(2,y,z))
14 #define Column(x,z)   (match(x,0,z) && match(x,1,z) && match(x,2,z))
15 #define Up(z)         (match(2,0,z) && match(1,1,z) && match(0,2,z))
16 #define Down(z)       (match(0,0,z) && match(1,1,z) && match(2,2,z))
17 #define horizontal(z) (Row(0,z) || Row(1,z) || Row(2,z))
18 #define vertical(z)   (Column(0,z) || Column(1,z) || Column(2,z))
19 #define diagonal(z)   (Up(z) || Down(z))
20 #define try(x,y,z)    b.r[x].c[y] == 0 -> b.r[x].c[y] = z
21
22 inline check_win(z) {
23      if
24      :: horizontal(z) || vertical(z) || diagonal(z) -> end_game: 0
25      :: else // continue the game
26      fi
27 }
28
29 inline place(z) {
30      if
31      :: try(0,0,z) :: try(0,1,z) :: try(0,2,z)
32      :: try(1,0,z) :: try(1,1,z) :: try(1,2,z)
33      :: try(2,0,z) :: try(2,1,z) :: try(2,2,z)
34      // if no moves are possible, it?s a draw
35      fi
36 }
37
38 init {
39      mtype symbol = cross;
40 end: do
41      :: atomic {
42           place(symbol);
43           c_code { now.abstract = abstract_value(); };
44           check_win(symbol);
45           symbol = (symbol == circle -> cross : circle);
46         }
47      od
48 }
```

Fig. 6 Spin model for the game of tic-tac-toe using an abstraction function

The only other issue that we now have to address is that with only the change above the model checker would see two representations of the same state: the concrete representation from before and the abstract representation that we just added. Clearly we will need both, since the concrete state determines at each point which moves are valid, and we need the abstract state to determine whether we are exploring a previously unseen board state. The method we can use in Spin to accomplish this exploits the fact that depth-first search uses two different data structures: the set of open states D (also known as the search stack) and the set of visited states V (also known as the state space). The nested depth-first search with abstraction illustrated in Fig. 4 stores the abstract states in V and the concrete states in D. The only thing

we need to be able to do, then, is to effect that the concrete state of the game board, `square b` in Fig. 5, is not stored in V but in D.

Hiding the concrete state of `square b` from both the state vector and the stack is done with declaration primitive `hidden`. But this accomplishes too much, since we do want to track the state of `square b` in search stack D. We can do this with the help of a `c_track` statement.

The first argument to `c_track` is a pointer to the data structure that we want to track. Because `square b` is declared `hidden`, this is a simple reference to `b`, without the need for a prefix (i.e., it is not part of the state vector). The second argument specifies the size of the object being tracked, which in this case is simply the size of data structure `square`. For the third argument we use the keyword `UnMatched`, which indicates that indeed the value of this object is not part of the system state as stored in the state space. The only alternative option for the third argument would be the keyword `Matched`, which however would completely undo the effect of declaration prefix `hidden`. In this case it seems redundant to have to use both `hidden` in the data declaration and `UnMatched` in the `c_track` statement, but note that in general `c_track` statements can be used to track the state of any data object, including those that are declared in C code external to the model checker where we often need the capability to consider this external data to be part of the state as stored in state space V.

We can now complete the model by including the abstraction function itself, which is written in C here, into the model. We can do this with an embedded `c_code` statement, which in Fig. 6 is placed immediately following the declaration of the new variable abstract.

The revised model using the abstraction function reaches 765 states, down from the original 5,528 states, matching the exact number of uniquely different board positions [1]. The abstraction we have used is logically sound (satisfying conditions (1) and (2) above), which means that the abstract model can prove precisely the same properties as the non-abstract version.

5.7 Model-Driven Verification

In the process of constructing the abstract version of the tic-tac-toe model, we have also seen how an explicit-state model checker can be used to track data that is declared outside the model itself, in C code, and how it can call an external C function to perform a state transition (e.g., one that modifies the externally declared tracked data). This is a powerful concept that makes it possible to use abstraction in full LTL verification of implementation level code.

In this *model-driven verification* approach, the non-determinism that is supported by the model checker can be used to drive an application into all relevant states, as reflected in the values of the tracked variables. An abstraction function is again used to compute an abstract representation of the data being tracked and stores it in canonical form. A linked list data structure, for example, consists of both the data

that is stored in the list elements and the pointers that connect the list elements. An abstraction function can collect just the list data elements into a simple array, preserving their order while abstracting from all pointer values. Note that the pointer values have no significance other than to fix the order of the list elements. Two linked lists with the same contents stored in two different places in memory would differ as concrete objects (because the pointer values differ), but they can trivially be recognized as equivalent under this abstraction. Arbitrary LTL properties (both safety and liveness) can be proven in this way with explicit-state model-checking techniques for even large applications, while using abstraction. Any counter-example generated is immediately also a concrete counter-example, because the concrete values for the full error trace are available on the search stack.

5.8 Incomplete Storage

One of the benefits of the explicit-state model-checking procedure is that we can store a different representation of system states in state space V than we do on search stack D. When abstraction is used, the state space can be restricted to storing only abstract values, which in general take up less space than the concrete values they represent. But we can also use other storage techniques to exploit this capability. We can, for instance, use lossless compression techniques. An attractive aspect here is that the compression need not be reversible.

Given that we have considerable freedom in choosing a storage strategy, it is natural to ask at this point what would happen if we permitted the state compression technique to be lossy. If $f(s)$ represents the compression function for state s, then a lossless compression function has the property that

$$\forall s, s',\ f(s) = f\left(s'\right) \Rightarrow s = s'. \tag{3}$$

We lose this property if $f()$ is allowed to be lossy.

5.8.1 Bitstate Hashing and Bloom Filters

Consider the case where we use f to compute an N-bit hash function of the bit-representation of the state-descriptor. No matter what the size of s is, $f(s)$ then always returns an N-bit representation of it. To store the state now requires setting just 1 bit in a 2^N-bit memory arena. For $N = 32$, for instance, 2^{32} bits or 512 Mbyte suffices to store all reachable states explicitly after this type of possibly lossy compression. There are several reasons why this can be attractive. One reason is that it requires only a fixed amount of memory, independent of the actual problem size. A second reason is that the storage operation itself is very fast, given that it takes only one single-bit operation. But what are the possible effects of any hash collisions?

A hash collision occurs when two different states ($s \neq s'$) produce the same compressed value ($f(s) = f(s')$). If this happens, the model checker will conclude (Fig. 4) that the new state was previously visited, when in fact it was not. The search is truncated and some reachable system states can remain unvisited. The search becomes incomplete. Despite this incompleteness, one very important property of the model-checking procedure is preserved though: when a counter-example is found it will always be an accurate counter-example to the property being checked. Counter-examples can be missed, but those that are found are uncorrupted. In considering this limitation it is important to realize that also a search that performs lossless storage (with or without abstraction) can become incomplete. A computer always has only finite memory and when all available memory has been used the search will come to a stop. If the fraction of all reachable states that can be explored with the hash method is larger than the fraction that can be explored with a lossless storage technique, then the lossy technique is clearly preferred. A small example can illustrate this.

Consider a verification model with a reachable state space of one million states of 1024 bytes each. Storing all states explicitly would require 1 Gbyte of memory. If we have only 128 Mbyte of memory available then we can explore no more than approximately 10% of all states in a single run. We can use the same 128 Mbyte of available memory, though, as a hash arena and use a hash function that compresses each state to 27 bits which can be used directly as bit addresses in the 128-Mbyte arena. This hash arena would have room to store up to 10^9 states, of which we will use just 0.1%.

With a good quality hash function we therefore statistically have a good probability to explore *all* of the 10^6 reachable states, giving us full problem coverage in most cases. Even though this makes this search method (implemented in most explicit-state model checkers as a *bitstate* or *supertrace* search mode) attractive, we have to trade certainty of problem coverage for a statistical expectation.

The search method we have described was introduced in 1987 as the *bitstate hashing* or *supertrace* method. Over the years, it has proven to be a very effective tool in handling large verification models in applications of explicit-state model checking [14]. The theoretical justification for the method turns out to be very similar to that of a storage method that was first described by Burton Bloom in 1970 [4]. The algorithm, or variations of it, e.g., the hash-compact method [32], have been implemented in almost all explicit-state model-checking systems.

5.9 Extensions

The basic explicit-state model-checking procedure we have described so far can be extended in many different ways to support different types of optimizations. The specific type of optimization to be chosen will generally depend on which critical resource use needs to be reduced. Two relevant extensions to explicit-state model checking that have been studied recently are, for instance,

- Context-bounded model checking: To search for those counter-examples that contain the fewest number of context switches. The enforcement of this constraint in a model-checking procedure generally increases run-time and memory use [19].
- Multi-core model-checking algorithms: The objective of these search algorithms is to reduce the run-time requirements of a model-checking run by leveraging the computational power of larger numbers of processing cores (or CPUs in a grid or cloud network) [3, 16–18, 20].

5.10 Synopsis

In this chapter we have presented an overview of explicit-state logic model-checking procedures, specifically those based on an automata-theoretic framework. We have illustrated how this approach supports the use of powerful abstraction techniques and, through the use of embedded C code to specify parts of a logic model, how it can be combined with methods for the direct verification of implementation level artifacts.

References

1. The game of tic-tac-toe. http://f2.org/maths/ttt.html
2. Avrunin, G., Corbett, J., Dwyer, M., Pasareanu, C., Siegel, S.: Benchmarking finite-state verifiers. Int. J. Softw. Tools Technol. Transf. **2**(4), 317–320 (2000)
3. Barnet, J., Brim, L., Rockai, P.: DiVinE multi-core, a parallel LTL model checker. In: Liu, Z., Ravn, A.P. (eds.) Intl. Symp. Automated Technology for Verification and Analysis (ATVA). LNCS, vol. 5799, pp. 234–239. Springer, Heidelberg (2009)
4. Bloom, B.: Spacetime trade-offs in hash coding with allowable errors. Commun. ACM **13**(7), 422–426 (1970)
5. Bosnacki, D.: Enhancing state space reduction techniques for model checking. Ph.D. thesis, Eindhoven University of Technology (2001)
6. Clarke, E., Emerson, E., Jha, S., Sistla, A.: Symmetry reduction in model checking. In: Hu, A.J., Vardi, M.Y. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 1427, pp. 147–158. Springer, Heidelberg (1998)
7. Corbett, J.: Evaluating deadlock detection methods for concurrent software. Trans. Softw. Eng. **22**(3), 161–180 (1996)
8. Courcoubetis, C., Vardi, M., Wolper, P., Yannakakis, M.: Memory efficient algorithms for the verification of temporal properties. Form. Methods Syst. Des. **1**(2–3), 275–288 (1992)
9. Dams, D., Grumberg, O.: Abstraction and abstraction refinement. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
10. Etessami, K., Holzmann, G.: Optimizing Büchi automata. In: Palamidessi, C. (ed.) Proc. 11th Intl. Conf. on Concurrency Theory (CONCUR). LNCS, vol. 1877, pp. 153–167. Springer, Heidelberg (2000)
11. Gerth, R., Peled, D., Vardi, M., Wolper, P.: Simple on-the-fly automatic verification of linear temporal logic. In: Proc. of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification XV, pp. 3–18. Chapman & Hall, London (1996)
12. Hajek, J.: Automatically verified data transfer protocols. In: Intl. Conf. on Computer Communication (ICCC), pp. 749–756 (1978)

13. Holzmann, G.: PAN: a protocol specification analyzer. Tech. Rep. TM81-11271-5, AT&T Bell Laboratories, (1981)
14. Holzmann, G.: An improved reachability analysis technique. Softw. Pract. Exp. **18**(2), 137–161 (1988)
15. Holzmann, G.: The Spin Model Checker: Primer and Reference Manual. Addison-Wesley, Reading (2004)
16. Holzmann, G.: Parallelizing the Spin model checker. In: Donaldson, A.F., Parker, D. (eds.) Intl. Workshop on Model Checking Software (SPIN). LNCS, vol. 7385, pp. 155–171. Springer, Heidelberg (2012)
17. Holzmann, G.: Proving properties of concurrent programs. In: Bartocci, E., Ramakrishnan, C.R. (eds.) Intl. Symposium on Model Checking of Software (SPIN). LNCS, vol. 7976, pp. 18–23. Springer, Heidelberg (2013)
18. Holzmann, G., Bosnacki, D.: The design of a multi-core extension of the Spin model checker. Trans. Softw. Eng. **33**(10), 659–674 (2007)
19. Holzmann, G., Florian, M.: Model checking with bounded context switching. Form. Asp. Comput. **23**(3), 365–389 (2011)
20. Holzmann, G., Joshi, R., Gorce, A.: Swarm verification techniques. Trans. Softw. Eng. **37**(6), 845–857 (2011)
21. Holzmann, G., Peled, D.: An improvement in formal verification. In: Proc. of the 7th IFIP WG6.1 International Conference on Formal Description Techniques VII, pp. 197–211. Chapman & Hall, London (1995)
22. Holzmann, G., Peled, D., Yannakakis, M.: On nested depth first search. In: Grégoire, J.C., Holzmann, G., Peled, D. (eds.) The Spin Verification System. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 32, pp. 23–32. DIMACS/AMS, Providence (1996)
23. Ip, C., Dill, D.: Better verification through symmetry. Form. Methods Syst. Des. **9**(1–2), 41–75 (2006)
24. Park, D.: Concurrency and automata on infinite sequences. In: Deussen, P. (ed.) GI-Conference on Theoretical Computer Science. LNCS, vol. 104, pp. 167–183. Springer, Heidelberg (1981)
25. Peled, D.: Partial-order reduction. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
26. Pnueli, A.: The temporal logic of programs. In: Annual Symp. on Foundations of Computer Science (FOCS), pp. 46–57. IEEE, Piscataway (1977)
27. Tanenbaum, A.: Computer Networks, 1st edn. Prentice Hall, New York (1981)
28. Vardi, M., Wolper, P.: Reasoning about infinite computations. Inf. Comput. **115**(1), 1–37 (1994). Journal version of a conference paper first published in 1983
29. West, C.: General technique for communications protocol validation. IBM J. Res. Dev. **22**(3), 393–404 (1978)
30. West, C., Zafiropulo, P.: Automated validation of a communications protocol: the CCITT X.21 recommendation. IBM J. Res. Dev. **22**(1), 60–71 (1978)
31. Wolper, P.: Specifying interesting properties of programs in propositional temporal logic. In: Symp. on Principles of Programming Languages (POPL), pp. 184–193. ACM, New York (1986)
32. Wolper, P., Leroy, D.: Reliable hashing without collision detection. In: Courcoubetis, C. (ed.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 697, pp. 59–70. Springer, Heidelberg (1993)

Chapter 6
Partial-Order Reduction

Doron Peled

Abstract Partial order reduction methods help reduce the time and space required to automatically verify concurrent asynchronous systems based on commutativity between concurrently executed transitions. We describe partial order reduction for various specification formalisms, such as LTL, CTL, and process algebra.

6.1 Introduction

Model checking of concurrent systems is an intractable problem; this is manifested in the *state space explosion* problem, where the number of states of a concurrent system can grow exponentially with the number of processes. However, in some sense, many of the executions of the system are equivalent, containing the same information. The main observation in partial order reduction is that often the property to be checked does not distinguish between executions that only differ from each other in the order of independent (concurrent) transitions.

Interleaving semantics, which is commonly used in modeling and verifying concurrent systems, distinguishes between executions that differ only in the order of independent transitions; those transitions appear in different orders. Partial order semantics [25] puts an order on events, sometimes called "causal order", when one event must end before the other can begin, while independently occurring events appear unordered. The connection between these semantics is that a single partial order execution can represent multiple interleaving executions, which are the well-founded[1] linearizations of the partial order execution. It is quite natural to expect that the formal specification of a system often cannot distinguish between interleaving executions that are linearizations of the same partial order execution. Hence, the idea behind partial order reduction is to construct and use a smaller state space that includes *representatives* for the equivalence classes of the executions [20]. In particular, a counterexample for the checked property must be present in the reduced state

[1] That is, each event can be preceded by finitely many events.

D. Peled (✉)
Bar Ilan University, Ramat Gan, Israel
e-mail: doron.peled@gmail.com

E.M. Clarke et al. (eds.), *Handbook of Model Checking*,
DOI 10.1007/978-3-319-10575-8_6

space exactly when one exists in the full state space. The *reduced* state space is constructed directly, without first constructing the full one. Ideally, it has a substantially smaller number of states.

Although the above intuitive explanation stands behind the name *partial order reduction*, in practice it is not always the case that this connection, between the interleaving and the partial order semantics, is precisely the one that is used for the reduction. In retrospect, *commutativity-based reduction* is perhaps a better name than the historical one: partial order reduction.

6.2 Partial Order Reduction

Several methods for reducing the state space used for temporal verification were developed around the beginning of the 1990s. The versions of the method called *ample sets* [20, 29], *stubborn sets* [33], and *persistent sets* [13] were developed independently, but with varying degrees of influence on each other, and sometimes in collaboration [12, 22]. The notions that will be used here are based on ample sets, but the other variations will also be described.

A *finite transition system* is a fivetuple (S, ι, T, AP, L) where

- S is a finite set of *states*,
- $\iota \in S$ is an *initial state*,
- T is a finite set of *transitions* such that each transition $\alpha \in T$ is a partial function $\alpha : S \mapsto S$,
- AP is a finite set of *propositions*, and
- $L : S \mapsto 2^{AP}$ is an *assignment function*.

A transition $\alpha \in T$ is *enabled* from a state s if $\alpha(s)$ is defined. That is, α can be applied to s, obtaining some successor state s'. Denote by $enabled(s)$ the set of transitions that are enabled from s.

An *execution* is a maximal alternating sequence $s_0\alpha_0 s_1\alpha_1 \ldots$ of states and transitions such that $s_0 = \iota$, and for each $i \geq 0$, $s_{i+1} = \alpha_i(s_i)$. We assume that an execution is always infinite. If the execution is finite, as there is no enabled transition that can extend it, we have reached termination or deadlock. The difference between termination and deadlock depends only on whether the finite sequence is terminated in a desired state, or prematurely, respectively. To handle just one type of sequence (infinite), we can convert a finite execution (i.e., either terminated or deadlocked) into an infinite one by repeating its last state indefinitely, through a new transition that is enabled when all the others are disabled, and does not change the state. For each execution ξ we can define the following sequences:

- the states sequence $st(\xi) = s_0 s_1 s_2 \ldots$;
- the transitions sequence $tr(\xi) = \alpha_0\alpha_1\alpha_2 \ldots$;
- the propositional sequence $pr(\xi) = L(s_0)L(s_1)L(s_2) \ldots$.

A *segment* is a finite or infinite contiguous part of an execution.

Explicit states-based model-checking techniques (including automata-based algorithms) perform a search, often a Depth-First Search (DFS), to explore the state space of the transition system. Then, some algorithms are applied to the state space. In practice, these algorithms are usually applied to the state space on the fly *during* its construction, rather than first completing the search and then performing the analysis.

Definition 1 An *independence relation* $I \subseteq T \times T$ is a symmetric and antireflexive relation on transitions. For each pair of independent transitions $(\alpha, \beta) \in I$ and state $s \in S$ such that $\alpha, \beta \in enabled(s)$, the following hold:

$\alpha \in enabled(\beta(s))$. [Independent transitions cannot disable each other.]
$\alpha(\beta(s)) = \beta(\alpha(s))$. [Executing two enabled independent transitions in any order results in the same global state.]

The *dependency* relation $D = (T \times T) \setminus I$ is the complement of the independence relation. For example, two local transitions in different processes (threads) that do not use any global variables are independent of each other. Asynchronous *send* and *receive* transitions (in different processes) are also independent. A variant of this definition [21] replaces the fixed independence relation with a relation that is sensitive also to the states from which the transitions are taken. This refinement can improve the reduction [13, 21].

Definition 2 A transition $\alpha \in T$ is *invisible* if for each $s, s' \in S$ such that $s' = \alpha(s)$, $L(s) = L(s')$.

Definition 3 The *stutter removal operator* $\natural$ applied to a propositional sequence ρ results in the sequence $\natural(\rho)$ where each maximal block of consecutive repetition of identical labeling by propositions is replaced with a single occurrence of that labeling. Two propositional sequences σ, ρ are *equivalent up to stuttering* (or *stuttering equivalent*) if $\natural(\sigma) = \natural(\rho)$. This is denoted $\sigma \equiv_{\natural} \rho$.

For example, if AP={p, q}, the finite sequences $\sigma = (p)(p, q)(p, q)(q)(q)(p, q)$ and $\rho = (p)(p)(p, q)(p, q)(p, q)(q)(p, q)$ are stuttering equivalent since $\natural(\sigma) = \natural(\rho) = (p)(p, q)(q)(p, q)$.

Typical specification for software (and asynchronous hardware) does not distinguish between two executions that are equivalent up to stuttering. In particular, Linear Temporal Logic (LTL) cannot distinguish between two stuttering equivalent sequences when disallowing the nexttime operator ('$\bigcirc$'). Lamport argued that specification *should* be closed under stuttering equivalence [24]. LTL without the nexttime operator corresponds exactly to stuttering-closed first-order monadic logic [31]. See also [32] for an algorithm for checking whether an LTL property *with* the nexttime operator is stuttering-closed.

The partial order reduction generates a reduced state space such that, for each execution in the full state space, there is a stuttering equivalent sequence in the

reduced one. When a pair of independent transitions α, β are enabled at s and *at most one* of them is visible, we have one of the following cases:

α is invisible. $L(s) = L(\alpha(s))$, $L(\beta(s)) = L(\alpha(\beta(s)))$.
β is invisible. $L(s) = L(\beta(s))$, $L(\alpha(s)) = L(\beta(\alpha(s)))$.
α, β are invisible. $L(s) = L(\alpha(s)) = L(\beta(s)) = L(\alpha(\beta(s)))$.

In each of these three cases, there is at most one change in L when going from s to $r = \alpha(\beta(s))$. The difference between executing α before β or β before α in the first two cases amounts to stuttering.

Reduction for LTL

In this section, the basic ample set partial order reduction, and some other variants, are described. The basic partial order reduction algorithms are usually described as variants of the classical DFS algorithm. The procedure `hash` is a standard hashing of its parameter in a hash table. One can check whether that value was hashed ($hashed(s)$), i.e., was visited already. The simple DFS algorithm is as follows.

```
proc Dfs(s);
     local variable s';
     hash(s);
     foreach α ∈ enabled(s) do
             Let s' be such that s --α--> s'
             if ¬hashed(s') then Dfs(s');
end Dfs;
```

We later use the fact that during DFS, reaching a state that is already on the search stack means closing a cycle.

Partial order reduction (and some other reduction techniques such as symmetry reduction) is based on calculating a subset $ample(s)$ of the enabled transitions $enabled(s)$ from each state that is expanded. Then, only the transitions in $ample(s)$ are taken from the current state, instead of all the transitions in $enabled(s)$:

```
proc Dfs(s);
     local variable s';
     hash(s);
     Calculate ample(s)
     foreach α ∈ ample(s) do
             Let s' be such that s --α--> s'
             if ¬hashed(s') then Dfs(s');
end Dfs;
```

If $ample(s) = enabled(s)$, we say that s is *fully expanded*. Choosing the subset $ample(s)$ of enabled transitions is done according to some conditions that are described below. These conditions are designed to guarantee that the existence of a counterexample for the checked property must be preserved between the full state space and the reduced one. In fact, these conditions guarantee an even stronger property, namely that the generated state space includes at least one representative that

is stuttering equivalent to each execution in the full state space. For a formal proof of that, one can refer to, e.g., [5].

C1 [13, 20, 29, 33] For every finite consecutive segment of an execution, starting from the state s, a transition that is dependent on a transition in $ample(s)$ cannot be executed before a transition from $ample(s)$.

To understand condition **C1**, consider a suffix σ of an execution, starting at s. There are two cases:

Case 1. For some $\alpha \in ample(s)$, α is the first transition from $ample(s)$ that appears in σ. Then, α is independent of all the transitions that precede it in σ. By applying Definition 1 repeatedly, all the transitions in σ prior to α can be commuted with α, obtaining a segment σ'.

Case 2. Any $\alpha \in ample(s)$ is independent of all the transitions in σ. By Definition 1, one can form a segment σ' by executing α and then the transitions in σ in that order.

Condition **C1** is quite abstract. Implementing it needs to take into account the particular mode of execution, e.g., shared variables, asynchronous, or synchronous message passing [12, 13, 17, 35]. For example, such a transition α can be local to some process.

An alternative definition for Condition **C1** appeared earlier in the context of deductive verification. This is part of a proof system that is based on using representatives [20], rather than a model-checking algorithm. There, the set of transitions of the verified system is decomposed into three parts: (1) a set of transitions that can be executed to generate the representative successors, (2) a set of disabled transitions that cannot become enabled without the execution of a transition from the first set, and (3) the rest of the transitions, which must be independent of all the transitions in the first set.

A similar definition to the one in [20], applied to model checking rather than a proof system, appears in [33]: a *stubborn set* contains a set of currently enabled transitions (similar to the role of (1) in the previous paragraph), and currently disabled transitions (similar to the role of (2)) that can become enabled only by the execution of the enabled transitions in the stubborn set. The rest of the transitions must be independent with respect to the enabled transitions in the stubborn set. Furthermore, in order to make the disabled transitions in the stubborn set become enabled, one needs to identify variables whose value must first be changed by transitions that are in the stubborn set.

To ensure that $pr(\sigma)$ and $pr(\sigma')$ will be stuttering equivalent (for both of the above cases) we enforce the following condition:

C2 [30] If s is not fully expanded then all of the transitions in $ample(s)$ are invisible.

To see the combined effect of **C1** and **C2**, consider a suffix of an execution $\sigma_1 = \beta_0\beta_1\beta_2\ldots$, which is executed from a state s, where $\beta_0 \notin ample(s)$. Then, according to **Case 1** above, there is some $\beta_j \in ample(s)$ such that $\sigma_2 =$

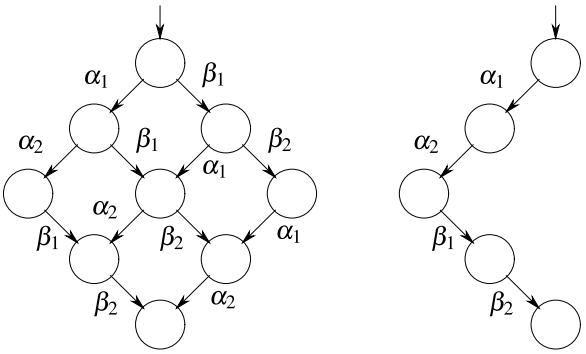


Fig. 1 Reduction with ample sets

$\beta_j\beta_0\beta_1\ldots\beta_{j-1}\beta_{j+1}\ldots$ is also a suffix of an execution from s, i.e., σ_2 is obtained from σ_1 by commuting β_j with all the transitions that precede it. Both sequences, σ_1 and σ_2, appended to any transition sequence from ι to s, form an execution of the checked system. According to **C2**, β_j is invisible, hence the corresponding propositional sequences are stuttering equivalent. Note that σ_1 is not in the reduced state space since $\beta_0 \notin ample(s)$. However, σ_2 may also be absent from the reduced state space due to later selections of ample sets; in this case, the same argument can be iterated starting from $\beta_j(s)$ to form a sequence that is stuttering equivalent to the original one in the reduced state space.

Handling **Case 2** is similar. Suppose no transition in $ample(s)$ appears in the suffix of an execution σ_1 that starts from s. Then any $\alpha \in ample(s)$ is independent of all of the transitions of σ_1. Consequently, $\alpha\beta_0\beta_1\beta_2\ldots$ is a suffix of an execution starting from s. Due to **C2** and the conditions on independence, we again obtain (together with a prefix from ι to s) a propositional sequence that is stuttering equivalent to the one not including α. Note that the sequence that starts with α is not a permutation of the original sequence, which is absent from the reduced state space. In this way, the partial order reduction deviates from providing a representative sequence for each set of well-founded linearizations of a partial order execution. Furthermore, we must guarantee that selecting β_0 (and subsequently β_1, β_2, and so forth) is not deferred indefinitely in the reduced state space in favor of independent transitions such as α in this case. This is guaranteed by imposing condition **C3**, which is described below.

To illustrate the reduction obtained using Conditions **C1** and **C2**, suppose, for example, that we have two parallel processes, one with two local transitions α_1 and α_2, executed sequentially, and the other also with two local transitions β_1 and β_2, also executed sequentially. The full state space of the system is shown on the left-hand side of Fig. 1. Applying the partial order reduction, one choice for an ample set from the initial state consists of the single transition α_1. Then, in the next state, we can select an ample set that consists of only α_2. Subsequently, only β_1 is enabled, and afterward β_2. This is shown on the right-hand side of Fig. 1.

Expanding $ample(s)$ from s instead of $enabled(s)$ defers the execution of a transition $\beta \in enabled(s) \setminus ample(s)$. (Notice that β remains enabled in any state $\alpha(s)$

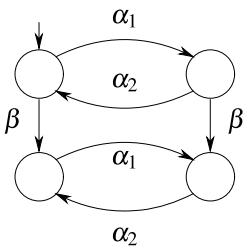


Fig. 2 The problem with loops

for $\alpha \in ample(s)$.) With only Conditions **C1** and **C2**, a transition can be deferred forever along a cycle. This may result in an execution that is not represented in the reduced state space by another stuttering equivalent execution, and can consequently lead to incorrect verification result. Figure 2 describes a full state space of a system with a loop α_1 then α_2 in one process, and independently, a transition β in a second process. Suppose that the ample set from each state consists of either α_1 or α_2; then the reduced state space includes only the upper cycle, never taking into account the execution of β.

The following condition guarantees that no transition is deferred forever.

C3 [29] If s is not fully expanded then no transition $\alpha \in ample(s)$ is on the search stack of the DFS.

This condition is easy to implement 'on the fly' during DFS.

There are different alternatives for condition **C3**, for example, Valmari [33] presents an algorithm that guarantees the following condition:

C3i For every cycle in the reduced state space there is at least one fully expanded node.

This is a more general condition, as it can easily be shown that **C3** implies **C3i**. Furthermore, it does not depend on a particular search strategy such as DFS, and hence can be used, e.g., with other search methods such as Breadth-First Search (BFS), and hence can be combined with methods that use Binary Decision Diagram (BDD) representation [1]. However, the algorithm implementing this condition in [33] is not completely compatible with on-the-fly model checking (see next section), as it requires storing the graph and correcting it to satisfy this condition.

Another variant of condition **C3** is based on the observation that each cycle of the state space must be the result of a combination of several local cycles of the separate concurrent processes. As a preparation for implementing this variant, the local structures of the processes can be analyzed before the global search begins, and at least one transition from each local cycle is preselected. This can be done by searching each local process separately, using DFS, and selecting each transition that causes that search to hit a state in the search stack of the local DFS. The selected transitions are called *sticky transitions*. The following condition is imposed:

C3ii [23] If s is not fully expanded then no transition $\alpha \in ample(s)$ is sticky.

With this new condition, the need to identify when a cycle is closed during the global state space exploration is eliminated.

Sticky transitions decrease the reduction and thus need to be minimized. To reduce the sticky transitions used, observe that there are some dependencies between local cycles of different processes. If one local cycle includes only local operations and receiving messages, another local cycle that includes sending messages must also be included to form a global cycle of the state space. Similarly, if one local cycle only decreases a variable, a local cycle of another process that increases it is also needed to complete a global cycle. Thus, local cycles that change some variable in a monotonic way can be exempt from the search for sticky transitions, but at the same time we must not exempt cycles that change that variable in the opposite polarity.

On-the-Fly Model Checking

In practice, model checking does not include a separate stage where the full or reduced state space is first generated before it is analyzed. The analysis of the state space can be performed during its construction. With the on-the-fly approach, if a counterexample is found, there is no need to further complete the construction of the state space. This observation can sometimes considerably reduce the memory size and time required for the verification. One way of performing on-the-fly model checking is to represent the state space as an automaton $\mathscr{A}$ recognizing the executions of the checked system. The checked property φ is also represented as an automaton $\mathscr{B}$. The automaton $\mathscr{B}$ accepts the sequences that do not satisfy φ (by a direct translation of $\neg\varphi$) [10, 36].

The intersection of $\mathscr{A}$ and $\mathscr{B}$ is an automaton recognizing executions of the system that *do not* satisfy the specification. Such executions exist iff the property φ is not satisfied by the system, and can be presented as counterexamples. Specifically, with on-the-fly model checking one can combine the following [30, 34]:

- the construction of an automaton $\mathscr{A}$ that corresponds to the *reduced* state space,
- the intersecting with the automaton $\mathscr{B}$, and
- checking for the emptiness of the intersection.

The conditions **C1**–**C3** guarantee that there is a counterexample for the checked property in the reduced state space exactly when there is a counterexample for the full state space (although the full state space may contain more counterexamples). The crucial change over the offline reduction presented in Sect. 6.2 is with respect to the cycle-closing Condition **C3**. A cycle found during an on-the-fly construction by returning to a state that is on the search stack is a cycle of the product automaton for $\mathscr{A} \cap \mathscr{B}$, rather than the cycles of the state space automaton $\mathscr{A}$.

Model checking for LTL properties often uses an efficient on-the-fly search strategy for an ultimately periodic sequence through an accepting state called *double depth-first search* [6, 18]. It is based on the fact that for finite state spaces, if there exists a counterexample for a checked LTL property, then there is in particular one that consists of a finite prefix and a finite iterative sequence. The first search then looks for the prefix and the second for the iterative part. The two searches are interleaved, where a second search starts each time we backtrack to an accepting state

found during the first search; this means that the two searches may close a cycle at different points. At the end of the search, the stack of the first DFS consists of the finite prefix, while the stack of the second DFS (in [18], combined with part of the stack of the first DFS) consists of the recurrent part. In addition to the possibility of the early termination of the search upon finding a counterexample, this search strategy avoids the need to explicitly keep the edges between the states.

To make the on-the-fly model checking work correctly with the partial order reduction, we can select a method that guarantees the conditions (in fact, **C3ii** rather than **C3**) for the state space of $\mathscr{A}$. From each current pair (s, q) of states of $\mathscr{A}$ and $\mathscr{B}$, respectively, we select an ample set from s, and pair each transition in it with the possible transitions of $\mathscr{B}$ from q. To do this, we use the sticky transition construction [23]. In this way, we cannot close a cycle of $\mathscr{A}$ without executing at least one sticky transition, and subsequently, without fully expanding at least one state on any cycle. This construction allows us to guarantee that the selection of ample sets is independent of whether s appears with a different $\mathscr{B}$ automaton component q or whether the search happens in the first or second DFS during double depth-first search.

Reduction for CTL

For branching temporal logics, we require that the partial order reduction generates a reduced state space that is *stuttering bisimilar* [3] to the full state space. Such a stuttering equivalence is defined between states of the full and reduced state spaces. Two states s and s' are related if the following conditions hold:

1. $L(s) = L(s')$,
2. for each infinite sequence σ starting from s there exists an infinite sequence σ' starting from s' such that σ and σ' can be partitioned into infinitely many finite blocks of consecutive states $B_0 B_1 \ldots$ and $B_0{}' B_1{}' \ldots$, respectively and the states in B_i are stuttering bisimilar to the states in $B_i{}'$ for each $i \geq 0$, and
3. symmetrically, for each sequence σ' from s' there exists a blockwise matching path σ from s.

It is shown in [3] that CTL and CTL* cannot distinguish between stuttering bisimilar structures. Reduction for CTL and CTL* is achieved by adding the following constraint:

C4 [9] If s is not fully expanded, then $ample(s)$ is a singleton.

Together with the other conditions, **C4** guarantees that when s is not fully expanded, s and its single successor, generated by the single transition in $ample(s)$, are stuttering bisimilar.

Reduction for Process Algebra

The focus in process algebras is on the branching structure of states and the execution of transitions rather than the appearance of states and their labeling with propositions. The models for various process algebras usually require the transitions rather than the states to be labeled. A transition labeled with τ is considered invisible, regardless of its effect on the state. Correctness in process algebras is usually based on simulation relations. Such relations associate corresponding pairs of states that have similar branching structure. Stuttering bisimulation was discussed above. Other relations are *branching bisimulation* [7, 11] and *weak bisimulation* [27].

The conditions **C1**–**C4** can be applied to produce a reduced structure that is branching bisimilar [9] and thus also weak bisimilar. One concern is that in process algebras transitions are often nondeterministic. For that, one can reformulate Condition **C4** as follows:

C4i [35] If s is not fully expanded, then $ample(s)$ is a singleton containing a deterministic transition.

Reducing Visibility

Experimental results [17] show that the reduction decreases rapidly when the number of visible transitions is increased. One way to reduce the effect of visibility on the partial order reduction is to let it dynamically decrease with the checked property [22]. We will illustrate this with an example. Suppose that the property to be checked is $\varphi = \Box(A \rightarrow \Box B)$. The negation of the property is $\neg\varphi = \Diamond(A \wedge \Diamond\neg B)$. Let $\mathscr{B}$ be an automaton for $\neg\varphi$. Once the model-checking search has reached a state of $\mathscr{B}$ where A holds, one can concentrate on checking that $\Diamond\neg B$ subsequently holds.

In this case, one may start with a set of transitions that are visible with respect to all the propositions that appear in the formula. In this case, the relevant propositions are $\{A, B\}$. Then once the property automaton $\mathscr{B}$ is left with a smaller goal, such as $\Diamond\neg B$, we can reduce the visible transitions to those that can affect the truth value of B. Those transitions that can only affect A can now be considered invisible. An LTL translation algorithm for an automaton $\mathscr{B}$ that can be used when reducing the set of visible transitions appears in [10]. This translation algorithm produces an automaton in which each state contains information about the subformulas that still needs to be satisfied.

6.3 Reducing Edges While Preserving States

Another kind of partial order reduction is aimed at reducing the edges traversed during a graph search. It may not be necessary to reduce the number of states reached, and, in fact, we may actually be required to reduce *only* the edges but *not* the states. Again, this is done based on principles of commutativity between transitions.

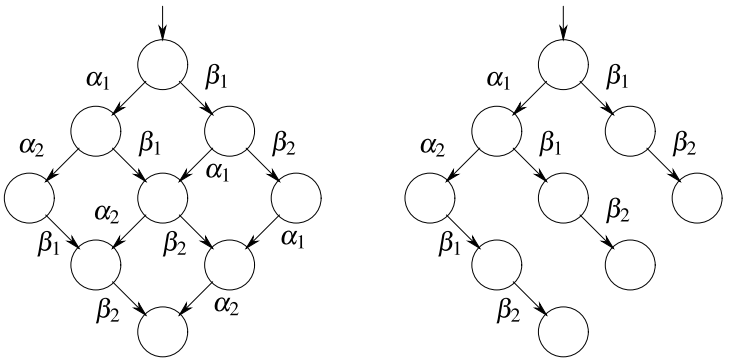


Fig. 3 Reduction with sleep sets or with TNF search

Sleep Sets

The sleep sets algorithm [14] keeps for each state s reached in the search a set $sleep(s)$ of transitions that will not be traversed from that set. The observation is that if these transitions were discovered from a previous state along the current path (which is in the search stack in DFS), then executing them from the current state will commute with the previous occurrences.

The basic algorithm from [14] appears below. We start the execution of the algorithm with the initial state ι and with an empty sleep set. When reaching a new state s from its predecessor, after each exploration of an edge marked with an enabled transition α from s, α is added to the sleep set of s. When passing to the successor state s' of s, upon executing a transition α, the sleep set of s is passed to s', except for all the transitions dependent on α ($dep(\alpha)$) are removed from the sleep set.

```
proc SleepSetsDfs(s,sleep);
     local variables s', current;
     current:=sleep;
     forall α ∉ sleep, s --α--> s' do
            begin
                  if s' not hashed then
                     SleepSetDfs(s',current \ dep(α));
                  current := current ∪ {α};
            end;
end SleepSetsDfs;
```

This algorithm explores all the states (but avoids a considerable number of edges) when the state space is acyclic [2]. Figure 3 shows an execution of the sleep set method over a system with the same transitions $\alpha_1; \alpha_2$ and $\beta_1; \beta_2$ in the different processes, as in Fig. 1. The left-hand side is the full state space again, while the right-hand side is the reduced state space. Assume that given that the transitions of both processes are enabled, those of the left process are explored first. In the first state, α_1 is explored first, and β_1 later (after backtracking from the successors in the α_1 direction). Now, after backtracking from the successors under α_1, α_1 is added to

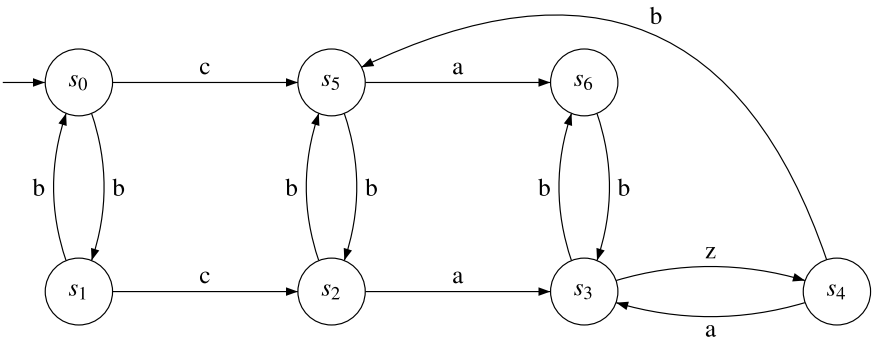


Fig. 4 A state space for which `TNF_Dfs` does not explore every state

the sleep set. Since α_1 and β_1 are independent, α_1 is not explored further from the successors under β_1.

Suppose now that there is a fixed total order $\ll$ among the edges, according to which they are traversed from each state (when enabled).

Figure 4 provides an example of a state space that is not fully explored by the sleep sets algorithm. The states, except s_6, are numbered in the order in which they are discovered. The node s_6 is not discovered. The transitions are $\{a, b, c, z\}$, with the ordering $a \ll b \ll c \ll z$. The independence relation in this example is the symmetric closure of $b\, I\, a$, $b\, I\, c$. Thus, z is dependent on every other letter a, b, c, and a, c are mutually dependent. The sleep sets are: $sleep(s_0) = sleep(s_1) = sleep(s_4) = \emptyset$, $sleep(s_2) = sleep(s_3) = \{b\}$, $sleep(s_5) = \{a\}$. The state s_6 can only be visited from s_3 with the edge b, and from s_5 with the edge a. The first case is eliminated since $b \in sleep(s_3)$. The second case is eliminated since $a \in sleep(s_5)$.

One way to increase the coverage is to store with each reached state its sleep set; then, if a state is reached again with a different sleep set that does not contain the old one, then the state is revisited [29]. The sleep set becomes the intersection of the new and the old sleep sets, and enabled transitions that are not in this intersection are reexplored [15, 29]. This algorithm can also be combined with an ample-set-style algorithm, to exploit the reductions of both methods. This requires revisiting and re-expanding already visited states and the storage of the sleep sets together with each state reached. In [16], reaching all the states is achieved by checking whether transitions in the sleep sets, explored from the current state, lead to a state on the search stack. If they do, they are removed from the sleep set. However, this means in fact that every edge in the full state space is explored in one way or another (but these latter edges can then be safely removed from the reduced graph).

Trace Normal Form

Let $\sigma,\ \rho \in \Sigma^*$. We write $\sigma \overset{1}{\equiv} \rho$ if and only if there exist strings $u, v \in \Sigma^*$ and letters $(a, b) \in I$ such that $\sigma = uabv$, $\rho = ubav$. That is, $\sigma \overset{1}{\equiv} \rho$ if ρ is obtained from σ

(or vice versa) by transposing adjacent independent letters. Let $\equiv$ be the transitive closure of $\overset{1}{\equiv}$. It is not hard to see that $\equiv$ is an equivalence relation. It is often called *trace equivalence* [25], where a trace is an equivalence class.

For example, for $\Sigma = \{a, b\}$ and $I = \{(a, b), (b, a)\}$ we have $abbab \overset{1}{\equiv} ababb$ and $abbab \equiv bbbaa$. Notice that if the system has the diamond property and $u \equiv v$, then $s \xrightarrow{u} r$ if and only if $s \xrightarrow{v} r$.

We can extend $\ll$ to a lexicographic order on words in a standard way, i.e., by setting $v \ll vu$ and $vau \ll vbw$ for any $v, u, w \in \Sigma^*$ and any $a, b \in \Sigma$ such that $a \ll b$.

Definition 4 Let $w \in \Sigma^*$. Let $\tilde{w}$ denote the least word under the relation $\ll$ that is equivalent to w. If $w = \tilde{w}$, we say that w is in *trace normal form* (TNF) [28].

Consider now a DFS where an enabled transition α is allowed only when concatenating it to the path currently residing in the search stack results in a string that is in TNF. In order not to keep and check this entire path, it is sufficient to keep with each state in the stack a *summary*, which contains the relative order of the last occurrence of each transition. Thus, if an edge a that appears in the summary appears again, it is removed from its old position, and appended to the end of the summary. Now, when progressing with the search, we keep in the stack, together with the state, also the index (position) of the transition that was shifted to the end. In this way, we do not need to keep the entire summary in the stack. We keep updating the summary as we progress and can easily recover it (using this index) upon backtracking.

The search obtained in this way progresses exactly like the sleep sets algorithm that uses the same order $\ll$ for selecting transitions. Consequently, this algorithm will also miss the state s_6 in the example of Fig. 4 according to the same independence relation and order of transitions. On the other hand, using only traces in normal form during Breadth-First Search (BFS) would not miss any state [2].

We describe here an algorithm `TNF_Dfs`(s_0) that only explores paths labeled with words in trace normal form. This algorithm often provides a significant reduction in the size of stack needed. For acyclic state spaces, `TNF_Dfs`(s_0) explores all states. However, as explained above, this may not be the case in general.

Definition 5 A *summary* of a string σ is the total order $\prec_\sigma$ on the letters from $\alpha(\sigma)$ such that $a \prec_\sigma b$ iff the last occurrence of a in σ precedes the last occurrence of b in σ. That is, $\sigma = vaubw$, where $v \in \Sigma^*$, $u \in (\Sigma \setminus \{a\})^*$, $w \in (\Sigma \setminus \{a, b\})^*$.

To perform a reduced search that only considers strings in TNF, we store the summary in a global array `summary[1..n]`, where $n = |\Sigma|$. The variable `size` stores the number of different letters in the current string σ. We update the summary as we progress with the DFS, and recover the previous value when backtracking, i.e., the value of the summary is calculated on the fly and not stored with the state information in the hash table. The value of the summary is calculated on the fly through

the use of functions `normal()`, `update_sumr()`, and `recover_sumr()` defined later. This means that there is no need to save the value of the summary with the state information.

```
size:=0;
TNF_Dfs(ι)

proc TNF_Dfs(s)
     local variables s', i;
     hash(s);
     forall s -a-> s' in increasing order do
            if normal(a) and s' not hashed then
               i:=ord(a);
               update_sumr(i,a);
               TNF_Dfs(s');
               recover_sumr(i,a);
end TNF_Dfs;
```

In order to perform the update, we need to keep the last transition a that was executed, and its old location i (0 if it was not introduced yet) in the summary. The update is performed using the procedure `update_sumr`. It pushes all the elements from the ith location to the left, and puts a at the end of the summary. If a did not occur in the summary, then there is no need for the shift, but in this case the size of the summary is increased.

```
proc update_sumr(i, a);
     if i=0 then
        size:=size+1;
     else
        for j:=i+1 to size do
            summary[j-1]:=summary[j];
     summary[size]:=a;
end update_sumr;
```

The function `ord` is used to find the position of a letter a in the summary.

```
func ord(a);
     for i:=size backto 1 do
         if summary[i]=a then return(i);
     return(0);
end ord;
```

The procedure `recover_sumr` is used to recover the previous summary upon backtracking. It reverses the effect of `update_sumr` by shifting the vector elements indexed i (the original position of a) and higher to the right, and putting a in the ith place. If i is zero, then there is no need for shifting, but the size of the summary needs to be decremented.

```
proc recover_sumr(i, a);
     if i=0 then
        summary[size]:=blank;
        size:=size-1;
     else
        for j:=size-1 downto i do
```

```
            summary[j+1]:=summary[j];
        summary[i]:=a;
end recover_sumr;
```

The reduced DFS procedure TNF_Dfs(s_0) considers all transitions enabled at the current state. For each of them, it checks whether the current string augmented with this transition is in TNF. This is done through a call to the function normal, which checks the summary.

```
func normal(a);
     for j:=size backto 1 do
         b:=summary[j];
         if ¬ (a I b) then return(true);
         if a≪b then return(false);
     return(true);
end normal;
```

Edge Lean Algorithm

In order to obtain a reduction that preserves all the states of the original state space, yet reduces edges, we use the following definition.

Definition 6 Set $ubav \Longrightarrow_1 uabv$ if and only if $a\ I\ b$ and $a \ll b$, and let $\Longrightarrow$ be the transitive closure of $\Longrightarrow_1$. We say that a word $w \in \Sigma^*$ is *irreducible* if there exists no $w' \neq w$ such that $w \Longrightarrow w'$.

Thus, a word is irreducible if it cannot be transformed into a smaller word with respect to $\Longrightarrow$ by permutations of adjacent independent letters. We call a path ρ irreducible if its labeling $\ell(\rho)$ is an irreducible word. Observe that a prefix of an irreducible path is also irreducible. Note that if w is in TNF, then it is irreducible. However, the converse does not necessarily hold. Indeed, consider $a \ll b \ll c$, $a\ I\ b$, $b\ I\ c$ and $a\ D\ c$. Then $x = cab$ is irreducible, but $\tilde{x} = bca \equiv x$, and $\tilde{x} \ll x$. Hence x is not in TNF.

The EdgeLeanDfs algorithm [2] is based on depth-first search. It only explores paths whose labelings are irreducible. For this, it suffices to remember the last letter x seen along the current path, and not to extend this path with letter y whenever $x\ I\ y$ and $y \ll x$.

```
EdgeLeanDfs(ι,ϵ);

proc EdgeLeanDfs(s,prev);
     local variable s';
     hash(s);
     forall s -a-> s' where prev=ϵ or ¬(aIprev) or prev ≪a do
            if s' not hashed then EdgeLeanDfs(s', a);
end EdgeLeanDfs;
```

6.4 Conclusions

Partial order reduction methods are aimed at reducing the time and space needed to check for properties of systems. They are based on the observation that such systems contain a lot of commutativity, generated by interleaving of concurrently (independently) executed transitions. As the specification is often insensitive to such order, one can exploit such a reduced state space to improve the efficiency of model checking. There are different techniques for achieving the reduction. Ample-set-type algorithms estimate a subset of transitions that are sufficient from the current state based on the current state and some structural properties of the system (e.g., the type of enabled transitions, be it local, asynchronous communication, etc.) and the nature of the search (e.g., when a cycle is closed during depth-first search). Sleep-set-type algorithms use some summary information about the history of the search so far to avoid looking at edges that were explored in executions equivalent up to commutativity.

Partial order reduction is most successfully applied to verification of software and asynchronous hardware. It is implemented mostly with state-space-based model checking. However, it can also be applied to symbolic model checking [19], by changing the search strategy (most notably, condition **C3**) to apply to breadth-first search [1, 4].

Another approach that uses partial order in verification of systems is called *unfolding* [26]. This approach, which was suggested by McMillan [26] is based on building the partial order representing the executions directly, rather than on a reduced set of representatives. The structure that is constructed is related to Winskel's *events structures* [37], which represent local events together with the causal order among them. In addition, the structure explicitly represents branching due to nondeterministic choice. The unfolding method avoids in the first place constructing executions (linearizations) that are equivalent up to commutativity. One needs to be careful about repeated branching, in particular when modeling programs that branch and later, the executions in the two branches meet, then branch again. A repetition of this behavior can enlarge the state space, due to duplication of successors that result from nondeterministic choice, making the unfolding structure much larger than the full interleaving state space. A solution to this, along with a comprehensive description of unfolding techniques, appears in [8].

References

1. Alur, R., Brayton, R., Henzinger, T., Qadeer, S., Rajamani, S.: Partial order reduction in symbolic state space exploration. In: Grumberg, O. (ed.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 1254, pp. 340–351. Springer, Heidelberg (1997)
2. Bosnacki, D., Elkind, E., Genest, B., Peled, D.: On commutativity based edge lean search. In: Arge, L., Cachin, C., Jurdzinski, T., Tarlecki, A. (eds.) Intl. Colloquium on Automata, Languages and Programming (ICALP). LNCS, vol. 4596, pp. 158–170. Springer, Heidelberg (2007)

3. Browne, M., Clarke, E., Grumberg, O.: Characterizing finite Kripke structures in propositional temporal logic. Theor. Comput. Sci. **59**(1–2), 115–131 (1988)
4. Chou, C., Peled, D.: Verifying a model-checking algorithm. In: Margaria, T., Steffen, B. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 1055, pp. 241–257. Springer, Heidelberg (1996)
5. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (2000)
6. Courcoubetis, C., Vardi, M., Wolper, P., Yannakakis, M.: Memory-efficient algorithms for the verification of temporal properties. Form. Methods Syst. Des. **1**(2–3), 275–288 (1992)
7. De Nicola, R., Vaandrager, F.: Three logics for branching bisimulation. In: Symp. on Logic in Computer Science, vol. LICS, pp. 118–129. IEEE, Piscataway (1990)
8. Esparza, J., Heljanko, K.: Unfoldings—a partial-order approach to model checking. EATCS Monographs in Theoretical Computer Science. Springer, Heidelberg (2008)
9. Gerth, R., Kuiper, R., Penczek, W., Peled, D.: A partial order approach to branching time logic model checking. In: Israel Symp. on the Theory of Computing and Systems (ISTCS), pp. 130–139. IEEE, Piscataway (1995). Full version in Information and Computation **150**(2), 132–152 (1999)
10. Gerth, R., Peled, D., Vardi, M., Wolper, P.: Simple on-the-fly automatic verification of linear temporal logic. In: Dembinski, P., Sredniawa, M. (eds.) Intl. Symp. on Protocol Specification, Testing and Verification (PSTV). IFIP Conference Proceedings, vol. 38, pp. 3–18. Chapman & Hall, London (1995)
11. van Glabbeek, R., Weijland, W.: Branching time and abstraction in bisimulation semantics. In: Ritter, G.X. (ed.) Information Processing 89, Proc. of the IFIP World Computer Congress, pp. 613–618. North-Holland/IFIP, Amsterdam (1989)
12. Godefroid, P., Peled, D., Staskauskas, M.: Using partial order methods in the formal validation of industrial concurrent programs. In: Zeil, S.J. (ed.) Intl. Symp. on Software Testing and Analysis (ISSTA). Software Engineering Notes, vol. 21, pp. 261–269. ACM Press, New York (1996)
13. Godefroid, P., Pirottin, D.: Refining dependencies improves partial order verification methods. In: Courcoubetis, C. (ed.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 697, pp. 438–449. Springer, Heidelberg (1993)
14. Godefroid, P., Wolper, P.: Using partial orders for the efficient verification of deadlock freedom and safety properties. In: Larsen, K.G., Skou, A. (eds.) Intl. Workshop on Computer-Aided Verification (CAV). LNCS, vol. 575, pp. 332–342. Springer, Heidelberg (1991)
15. Godefroid, P., Wolper, P.: Using partial orders for the efficient verification of deadlock freedom and safety properties. Form. Methods Syst. Des. **2**(2), 149–164 (1993)
16. Holzmann, G., Godefroid, P., Pirottin, D.: Coverage preserving reduction strategies for reachability analysis. In: Linn, R.J. Jr., Uyar, M.Ü. (eds.) Intl. Symp. on Protocol Specification, Testing and Verification (PSTV). IFIP Transactions, vol. C-8, pp. 349–363. North-Holland, Amsterdam (1992)
17. Holzmann, G., Peled, D.: An improvement in formal verification. In: Hogrefe, D., Leue, S. (eds.) Intl. Conf. on Formal Description Techniques (FORTE). IFIP Conference Proceedings, vol. 6, pp. 197–211. Chapman & Hall, London (1994)
18. Holzmann, G.J., Peled, D., Yannakakis, M.: On nested depth first search. In: Grégoire, J.C., Holzmann, G.J., Peled, D.A. (eds.) Workshop on the SPIN Verification System. DIMACS: Series in Discrete Mathematics and Theoretical Computer Science, vol. 32, pp. 23–32. AMS/DIMACS, Providence (1996)
19. Kahlon, V., Wang, C., Gupta, A.: Monotonic partial order reduction: an optimal symbolic partial order reduction technique. In: Bouajjani, A., Maler, O. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 5643, pp. 398–413. Springer, Heidelberg (2009)
20. Katz, S., Peled, D.: An efficient verification method for parallel and distributed programs. In: de Bakker, J.W., de Roever, W.P., Rozenberg, G. (eds.) REX Workshop on Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency. LNCS, vol. 354, pp. 489–507. Springer, Heidelberg (1988)

21. Katz, S., Peled, D.: Defining conditional independence using collapses. Theor. Comput. Sci. **101**(2), 337–359 (1992)
22. Kokkarinen, I., Peled, D., Valmari, A.: Relaxed visibility enhances partial order reduction. In: Grumberg, O. (ed.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 1254, pp. 328–339. Springer, Heidelberg (1997)
23. Kurshan, R., Levin, V., Minea, M., Peled, D., Yenigün, H.: Static partial order reduction. In: Steffen, B. (ed.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 1384, pp. 345–357. Springer, Heidelberg (1998)
24. Lamport, L.: What good is temporal logic. In: Mason, R. (ed.) Information Processing 83, Proc. of the World Computer Congress, pp. 657–668. North-Holland/IFIP, Amsterdam (1983)
25. Mazurkiewicz, A.: Trace theory. In: Brauer, W., Reisig, W., Rozenberg, G. (eds.) Petri Nets: Central Models and Their Properties, Advances in Petri Nets (APN). LNCS, vol. 255, pp. 279–324. Springer, Heidelberg (1986)
26. McMillan, K.L.: Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In: Gregor von Bochmann, D.K.P. (ed.) Intl. Workshop on Computer-Aided Verification (CAV). LNCS, vol. 663, pp. 164–177. Springer, Heidelberg (1992)
27. Milner, R.: A Calculus of Communicating Systems. LNCS, vol. 92. Springer, Heidelberg (1980)
28. Ochmanski, E.: Languages and automata. In: Diekert, V., Rozenberg, G. (eds.) The Book of Traces, pp. 167–204. World Scientific, Singapore (1995)
29. Peled, D.: All from one, one for all, on model-checking using representatives. In: Courcoubetis, C. (ed.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 697, pp. 409–423. Springer, Heidelberg (1993)
30. Peled, D.: Combining partial order reductions with on-the-fly model-checking. Form. Methods Syst. Des. **8**(1), 39–64 (1996)
31. Peled, D., Wilke, T.: Stutter-invariant temporal properties are expressible without the nexttime operator. Inf. Process. Lett. **63**(5), 243–246 (1997)
32. Peled, D., Wilke, T., Wolper, P.: An algorithmic approach for checking closure properties of ω-regular languages. In: Intl. Conf. on Concurrency Theory (CONCUR). LNCS, vol. 1119, pp. 596–610. Springer, Heidelberg (1996)
33. Valmari, A.: Stubborn sets for reduced state space generation. In: Rozenberg, G. (ed.) Intl. Conf. on Applications and Theory of Petri Nets (APN). LNCS, vol. 483, pp. 491–515. Springer, Heidelberg (1989)
34. Valmari, A.: On-the-fly verification with stubborn sets. In: Courcoubetis, C. (ed.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 697, pp. 397–408. Springer, Heidelberg (1993)
35. Valmari, A.: Stubborn set methods for process algebras. In: Peled, D.A., Pratt, V.R., Holzmann, G.J. (eds.) DIMACS Workshop on Partial Order Methods in Verification (POMIV). DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 29, pp. 213–231. AMS, Providence (1996)
36. Vardi, M., Wolper, P.: An automata-theoretic approach to automatic program verification. In: Symp. on Logic in Computer Science (LICS), pp. 322–331. IEEE, Piscataway (1986)
37. Winskel, G.: An introduction to event structures. In: de Bakker, J.W., de Roever, W.P., Rozenberg, G. (eds.) REX Workshop on Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency. LNCS, vol. 354, pp. 364–397. Springer, Heidelberg (1988)

Chapter 7
Binary Decision Diagrams

Randal E. Bryant

Abstract Binary decision diagrams provide a data structure for representing and manipulating Boolean functions in symbolic form. They have been especially effective as the algorithmic basis for symbolic model checkers. A binary decision diagram represents a Boolean function as a directed acyclic graph, corresponding to a compressed form of decision tree. Most commonly, an ordering constraint is imposed among the occurrences of decision variables in the graph, yielding *ordered* binary decision diagrams (OBDD). Representing all functions as OBDDs with a common variable ordering has the advantages that (1) there is a unique, reduced representation of any function, (2) there is a simple algorithm to reduce any OBDD to the unique form for that function, and (3) there is an associated set of algorithms to implement a wide variety of operations on Boolean functions represented as OBDDs. Recent work in this area has focused on generalizations to represent larger classes of functions, as well as on scaling implementations to handle larger and more complex problems.

7.1 Introduction

Ordered Binary Decision Diagrams (OBDDs) provide a symbolic representation of Boolean functions. They can serve as the underlying data structure to implement an abstract data type for creating, manipulating, and analyzing Boolean functions. OBDDs provide a uniform representation for operations to define simple functions and then construct representations of more complex functions via the operations of Boolean algebra, as well as function projection and composition. In the worst case, the OBDD representation of a function can be of size exponential in the number of function variables, but in practice they remain of tractable size for many applications.

OBDDs have been especially effective as a data structure for supporting symbolic model checking, starting with the very first implementations of tools for symbolically checking the properties of finite-state systems [10, 21, 25, 49]. By encoding

R.E. Bryant (✉)
Carnegie Mellon University, Pittsburgh, PA, USA
e-mail: randy.bryant@cs.cmu.edu

E.M. Clarke et al. (eds.), *Handbook of Model Checking*,
DOI 10.1007/978-3-319-10575-8_7

sets and relations as Boolean functions, the operations of model checking can be expressed as symbolic operations on Boolean functions, avoiding the need to explicitly enumerate any states or transitions.

In the spirit of viewing OBDDs as the basis for an abstract data type, we first define an Application Program Interface (API) for Boolean function manipulation, then the OBDD representation, and then how the API can be implemented with OBDDs. We describe some of the refinements commonly seen in OBDD implementations. We present some variants of OBDDs that have been devised to improve efficiency for some applications, as well as to extend the expressive power of OBDDs beyond Boolean functions. The many variants of OBDDs are sometimes referred to by the more general term *decision diagrams* (DDs). Many surveys of OBDDs and their generalizations have been published over the years [17, 18, 31]. Rather than providing a comprehensive survey, this chapter focuses on those aspects that are most relevant to model checking. We describe some efforts to improve the performance of OBDD programs, both to make them run faster and to enable them to handle larger and more complex applications. We conclude with a brief discussion on some relationships between OBDDs and Boolean satisfiability (SAT) solvers.

7.2 Terminology

Let $\mathbf{x}$ denote a vector of Boolean variables $x_1, x_2, \ldots, x_n$. We consider Boolean functions over these variables, which we write as $f(\mathbf{x})$ or simply f when the arguments are clear. Let $\mathbf{a}$ denote a vector of values $a_1, a_2, \ldots, a_n$, where each $a_i \in \{0, 1\}$. Then we write the *valuation* of function f applied to $\mathbf{a}$ as $f(\mathbf{a})$. Note the distinction between a function and its valuation: $f(\mathbf{x})$ is a function, while $f(\mathbf{a})$ is either 0 or 1.

Let $\mathbf{1}$ denote the function that always yields 1, and $\mathbf{0}$ the function that always yields 0.

We can define Boolean operations $\wedge$, $\vee$, $\oplus$, and $\neg$ over functions as yielding functions according to the Boolean operations on the underlying elements. So, for example, $f \wedge g$ is a function h such that $h(\mathbf{a}) = f(\mathbf{a}) \wedge g(\mathbf{a})$ for all $\mathbf{a}$.

For function f, variable x_i and binary value $b \in \{0, 1\}$, define a *restriction* of f as the function resulting when x_i is set to value b:

$$f|_{x_i \leftarrow b}(\mathbf{a}) = f(a_1, \ldots, a_{i-1}, b, a_{i+1}, \ldots, a_n).$$

The two restrictions of a function f with respect to a variable x_i are referred to as the *cofactors* of f with respect to x_i [13].

Given the two cofactors of f with respect to variable x_i, the function can be reconstructed as

$$f = (x_i \wedge f|_{x_i \leftarrow 1}) \vee (\neg x_i \wedge f|_{x_i \leftarrow 0}). \tag{1}$$

This identity is commonly referred to as the *Shannon expansion* of f with respect to x_i, although it was originally recognized by Boole [14].

Other useful operations on functions can be defined in terms of the restriction operation and the algebraic operations $\wedge$, $\vee$, $\oplus$, and $\neg$. Let f and g be functions over variables $\mathbf{x}$. The *composition* of f and g with respect to variable x_i, denoted $f|_{x_i \leftarrow g}$, is defined as the result of evaluating f with variable x_i replaced by the evaluation of g:

$$f|_{x_i \leftarrow g}(\mathbf{a}) = f\big(a_1, \ldots, a_{i-1}, g(a_1, \ldots, a_n), a_{i+1}, \ldots, a_n\big).$$

Composition can be expressed based on a variant of the Shannon expansion:

$$f|_{x_i \leftarrow g} = (g \wedge f|_{x_i \leftarrow 1}) \vee (\neg g \wedge f|_{x_i \leftarrow 0}). \tag{2}$$

Another class of operations involves eliminating one or more variables from a function via quantification. That is, we can define the operations $\forall x_i . f$ and $\exists x_i . f$ as:

$$\forall x_i . f = f|_{x_i \leftarrow 1} \wedge f|_{x_i \leftarrow 0} \tag{3}$$

$$\exists x_i . f = f|_{x_i \leftarrow 1} \vee f|_{x_i \leftarrow 0}. \tag{4}$$

By way of reference, the resolution step of the original Davis–Putnam (DP) Boolean satisfiability algorithm [28] can be seen to implement existential quantification for a function represented in clausal form. Their method is based on the principle that function f is satisfiable (i.e., $f(\mathbf{a}) = 1$ for some $\mathbf{a}$) if and only if $\exists x_i . f$ is satisfiable, for any variable x_i.

Quantification can be generalized to quantify over a set of variables $X \subseteq \{x_1, \ldots, x_n\}$. Existential quantification over a set of variables can defined recursively as

$$\exists \emptyset . f = f$$

$$\exists (x_i \cup X) . f = \exists x_i . (\exists X . f),$$

and the extension for universal quantification is defined similarly.

The ability of OBDDs to support variable quantification operations with reasonable efficiency is especially important for model checking, giving them an important advantage over Boolean satisfiability solvers. While deciding whether or not an ordinary Boolean formula is satisfiable is NP-hard, doing so for a quantified Boolean formula is PSPACE-complete [35].

Finally, we define the *relational product* operation, defined for functions $f(\mathbf{x})$, $g(\mathbf{x})$, and variables $X \subseteq \{x_1, \ldots, x_n\}$ as $\exists X . (f \wedge g)$. As is discussed in Chap. 8 [22], this operation is of core importance in symbolic model checking as the method to project a set of possible system states either forward (image) or backward (preimage) in time. Hence, it merits a specialized algorithm, as will be described in Sect. 7.5.

7.3 A Boolean Function API

As a way of defining an abstract interface for an OBDD-based Boolean function manipulation package, Fig. 1 lists a set of operations for creating and manipulating

Fig. 1 Basic operations for a Boolean function abstract data type

Operation	Result
Base functions	
CONST(b)	**1** (b = 1) or **0** (b = 0)
VAR(i)	x_i
Algebraic operations	
NOT(f)	$\neg f$
AND(f, g)	$f \wedge g$
OR(f, g)	$f \vee g$
XOR(f, g)	$f \oplus g$
Nonalgebraic operations	
RESTRICT(f, i, b)	$f\|_{x_i \leftarrow b}$
COMPOSE(f, i, g)	$f\|_{x_i \leftarrow g}$
EXISTS(f, I)	$\exists X_I . f$
FORALL(f, I)	$\forall X_I . f$
RELPROD(f, g, I)	$\exists X_I . (f \wedge g)$
Examining functions	
EQUAL(f, g)	$f = g$
EVAL(f, **a**)	$f(\mathbf{a})$
SATISFY(f)	some **a** such that $f(\mathbf{a}) = 1$
SATISFY-ALL(f)	$\{\mathbf{a} \mid f(\mathbf{a}) = 1\}$

Boolean functions and for examining their properties. In this figure f and g represent Boolean functions (represented by OBDDs), i is a variable index between 1 and n, b is either 0 or 1, and **a** is a vector of n 0s and 1s. For a set of indices $I \subseteq \{1, \ldots, n\}$, X_I denotes the corresponding set of variables $\{x_i \mid i \in I\}$. This figure is divided into several sections according to the different classes of operations.

The base operations generate the constant functions and functions corresponding to the individual variables. The algebraic operations have functions as arguments and generate new functions as results according to the operations $\wedge$, $\vee$, $\oplus$, and $\neg$. The nonalgebraic operations also have functions as arguments and as results, but they extend the functionality beyond those of Boolean algebra, implementing the operations of restriction, composition, quantification, and relational product.

The operations in the final set provide mechanisms to examine and test the properties of the generated Boolean functions. The EQUAL operation tests whether two functions are equivalent, yielding either true or false. As special cases, this operation can be used to test for tautology (compare to **1**) and (un)satisfiability (compare to **0**). The EVAL operation computes the value of a function for a specific set of argument values. For a satisfiable function, we can ask the program to generate some arbitrary satisfying solution (SATISFY) or have it enumerate all satisfying solutions (SATISFY-ALL.) The latter operation must be used with care, of course, since there can be as many as 2^n solutions.

The set of operations listed in Fig. 1 makes it possible to implement a wide variety of tasks involving the creation and manipulation of Boolean functions, including symbolic model checking. The overall strategy when working with OBDDs is to break a task down into a number of steps, where each step involves creating a new OBDD from previously computed ones. For example, a program can construct the OBDD representation of the function denoted by a Boolean expression by starting

with functions representing the expression variables. It then evaluates each operation in the expression using the corresponding algebraic operation on OBDDs until the representation of the overall expression is obtained.

As an illustration, suppose we are given the Boolean expression

$$(x_1 \wedge x_2 \wedge \neg x_3) \vee (\neg x_1 \wedge x_3). \tag{5}$$

We can create an OBDD for the function f denoted by this expression using a sequence of API operations:

$$\begin{aligned} f_1 &= \textsc{Var}(1) \\ f_2 &= \textsc{Var}(2) \\ f_3 &= \textsc{Var}(3) \\ f_4 &= \textsc{And}(f_1, f_2) \\ f_5 &= \textsc{Not}(f_3) \\ f_6 &= \textsc{And}(f_4, f_5) \\ f_7 &= \textsc{Not}(f_1) \\ f_8 &= \textsc{And}(f_7, f_3) \\ f &= \textsc{Or}(f_6, f_8) \end{aligned}$$

Similarly, given a combinational logic circuit, we can generate OBDD representations of the primary output functions by starting with OBDD representations of the primary input variables and then stepping through the network in topological order. Each step involves generating the OBDD for the function at the output of a gate according to the gate input functions and the gate operation.

7.4 OBDD Representation

A binary decision diagram (BDD) represents a Boolean function as an acyclic directed graph, with the nonterminal vertices labeled by Boolean variables and the leaf vertices labeled with the values 1 and 0 [1]. For nonterminal vertex v, its associated variable is denoted $var(v)$, while for leaf vertex v its associated value is denoted $val(v)$. Each nonterminal vertex v has two outgoing edges: $hi(v)$, corresponding to the case where its variable has value 1, and $lo(v)$, corresponding to the case where its variable has value 0. We refer to $hi(v)$ and $lo(v)$ as the *hi* and *lo children* of vertex v. The two leaves are referred to as the 1-*leaf* and the 0-*leaf*.

As an illustration, Fig. 2 shows a BDD representation of the function given by the expression in Eq. (5). In our figures, we show the arcs to the *lo* children as dashed lines and to the *hi* children as solid lines. To see the correspondence between the BDD and the Boolean expression, observe that there are only two paths from the root (vertex v_1) to the 1-leaf (vertex v_6): one through vertices v_2 and v_4, such that variables x_1, x_2, and x_3 have values 1, 1, and 0, and one through vertex v_3 such that variables x_1 and x_3 have values 0 and 1.

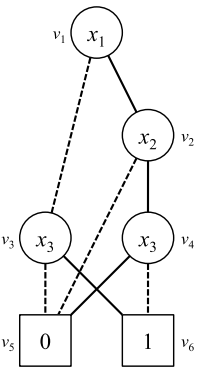


Fig. 2 OBDD representation of $(x_1 \wedge x_2 \wedge \neg x_3) \vee (\neg x_1 \wedge x_3)$

We can define the Boolean function represented by a BDD by associating a function f_v with each vertex v in the graph. For the two leaves, the associated values are **1** (1-leaf) and **0** (0-leaf). For nonterminal vertex v, the associated function is defined as

$$f_v = \big(var(v) \wedge f_{hi(v)}\big) \vee \big(\neg var(v) \wedge f_{lo(v)}\big). \tag{6}$$

We see here the close relation between the BDD representation of a function and the Shannon expansion; the two children of a vertex correspond to its two cofactors with respect to its associated variable. Every vertex in a BDD represents a Boolean function, but we designate one or more of these to be *root vertices*, representing Boolean functions that are referenced by the application program.

With *ordered* binary decision diagrams (OBDDs), we enforce an ordering rule on the variables associated with the graph vertices. For each vertex v having $var(v) = x_i$, and for vertex $u \in \{hi(v), lo(v)\}$ having $var(u) = x_j$, we must have $i < j$. For the rest of this chapter, we assume that all functions are represented as OBDDs with a common variable ordering. In the example BDD of Fig. 2, we see that the variable indices along all paths from the root to the leaves are in increasing order, and thus it is an OBDD.

We can define a *reduced* OBDD as one satisfying the rules:

1. There can be at most one leaf having a given value.
2. There can be no vertex v such that $hi(v) = lo(v)$.
3. There cannot be distinct nonterminal vertices u and v such that $var(u) = var(v)$, $hi(u) = hi(v)$, and $lo(u) = lo(v)$.

Given an arbitrary OBDD, we can convert it to reduced form by repeatedly applying transformations corresponding to these three rules:

1. If leaves u and v have $val(u) = val(v)$, then eliminate one of them and redirect all incoming edges to the other.
2. If vertex v has $lo(v) = hi(v)$, then eliminate vertex v and redirect all incoming edges to its child.
3. If vertices u and v have $var(u) = var(v)$, $hi(u) = hi(v)$, and $lo(u) = lo(v)$, then eliminate one of the vertices and redirect all incoming edges to the other one.

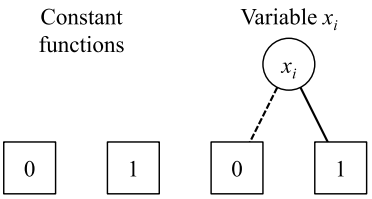


Fig. 3 OBDD representation for constant functions and variable x_i

By working from the leaves upward, and by employing sparse labeling techniques, this reduction can be performed in time linear in the size of the original graph [62]. The example OBDD of Fig. 2 is, in fact, a reduced OBDD.

Bryant showed that reduced OBDDs serve as a *canonical form* for Boolean functions [15]. That is, for a given variable ordering, every Boolean function over these variables has a unique (up to isomorphism) representation as a reduced OBDD.

There are two different conventions for representing multiple functions as OBDDs. In a *split* representation, each function has a separate OBDD, and each graph has a single root. In a *shared* representation [52], the reduction rules are applied across the entire set of functions, and so the entire collection of functions is represented as a single OBDD having multiple roots. The shared representation not only reduces the space required to represent a set of functions, it has the property that two represented functions are equal if and only if they are represented by the same vertex in the OBDD. That is, there cannot be two distinct vertices u and v such that $f_u = f_v$. This is sometimes referred to as a *strong canonical form*.

7.5 Implementing OBDD Operations

As Fig. 1 indicates, an OBDD software package must implement a number of operations. For those having a Boolean function as an argument or result, we denote this function by the root vertex in its OBDD representation. Thus, when describing OBDD algorithms, we define the operations in terms of vertex names, such as u and v, rather than abstract function names, such as f and g. As mentioned earlier, we can implement a set of Boolean functions as either a collection of separate OBDDs, each having a single root (split form), or as a single OBDD having multiple roots (shared form). In our presentation, we consider both approaches.

The base functions listed in Fig. 1 have simple representations as OBDDs, as shown in Fig. 3. Algorithms for the other operations follow a common framework based on depth-first traversals of the argument graphs. We present the *Apply* algorithm, a general method for implementing the binary Boolean algebraic operations, as an illustration.

The Apply algorithm has as arguments an operation *op* (equal to AND, OR, or XOR), as well as vertices u and v. The implementation, shown in Fig. 4, performs a depth-first traversal of the two argument graphs and generates a reduced OBDD from the bottom up as it returns from the recursive calls. The algorithm makes use of two data structures storing keys and values. The *computed cache* stores results

1. If the computed cache contains an entry with key $\langle op, u, v\rangle$, then return the associated value.
2. If one of the special cases shown in Fig. 5 applies, then return the specified value.
3. Recursively compute the two cofactors of the result as follows:
   a. Let $x_i = var(u)$, $x_j = var(v)$, and $k = \min(i, j)$.
   b. Compute u_1 and u_0 as $u_1 = hi(u)$ and $u_0 = lo(u)$ when $i = k$, and as $u_1 = u_0 = u$ when $i \neq k$.
   c. Compute v_1 and v_0 as $v_1 = hi(v)$ and $v_0 = lo(v)$ when $j = k$, and as $v_1 = v_0 = v$ when $j \neq k$.
   d. Compute $w_1 = \textsc{Apply}(op, u_1, v_1)$ and $w_0 = \textsc{Apply}(op, u_0, v_0)$.
4. Compute result vertex w:
   a. If $w_1 = w_0$, then $w = w_1$;
   b. else if the unique table contains an entry for key $\langle x_k, w_1, w_0\rangle$, then let w be the associated value;
   c. else create a new vertex w with $var(w) = x_k$, $hi(w) = w_1$, and $lo(w) = w_0$. Add an entry to the unique table with key $\langle x_k, w_1, w_0\rangle$ and value w.
5. Add an entry with key $\langle op, u, v\rangle$ and value w to the computed cache and return w.

Fig. 4 Recursive algorithm to compute $\textsc{Apply}(op, u, v)$

from previous invocations of the Apply operation, a process commonly referred to as *memoizing* [50]. Each invocation of Apply first checks this cache to determine whether a vertex corresponding to the given arguments has already been computed. As the name suggests, this data structure can be a cache where elements are evicted when space is needed, since the purpose of this data structure is purely to speed up the execution. The *unique table* contains an entry for every OBDD vertex, with a key encoding its variable and children. This table is used to ensure that duplicate vertices are not created. When using a split representation, this cache and table must be reinitialized for every invocation of Apply, while for a shared representation, the two data structures are maintained continuously.

Figure 5 shows cases where the recursive function implementing the Apply algorithm can terminate. As can be seen, the use of a shared representation enables additional terminal cases. In Sect. 7.6, we will discuss the use of *complement edges* to indicate the negation of a function. Their use enables even more terminal cases.

As the Apply algorithm illustrates, standard implementations of OBDD operations perform depth-first traversals of one or more argument graphs and generate reduced graphs as the recursions complete. The unique table is used to enforce reduction rules 1 and 3. The computed cache is used to stop the recursion when previously computed results are encountered. This cache can guarantee that the time required by the algorithm is bounded by the number of unique argument combinations. If the top-level arguments to APPLY have N_u and N_v vertices, respectively, then the total number of calls to APPLY is at most $N_u \times N_v$. Typical implementations of the computed cache and the unique table use hashing techniques, which can yield constant average time for each access, and thus the overall time complexity of APPLY is $O(N_u \times N_v)$.

Operation *op*	Condition	Restrictions	Result
AND	u, v are leaves		CONST($val(u) \wedge val(v)$)
AND	$u = \mathbf{0}$		CONST(0)
AND	$v = \mathbf{0}$		CONST(0)
AND	$u = \mathbf{1}$	S	v
AND	$v = \mathbf{1}$	S	u
AND	$u = v$	S	u
AND	$u =$ NOT(v)	S, C	CONST(0)
OR	u, v are leaves		CONST($val(u) \vee val(v)$)
OR	$u = \mathbf{1}$		CONST(1)
OR	$v = \mathbf{1}$		CONST(1)
OR	$u = \mathbf{0}$	S	v
OR	$v = \mathbf{0}$	S	u
OR	$u = v$	S	u
OR	$u =$ NOT(v)	S, C	CONST(1)
XOR	u, v are leaves		CONST($val(u) \oplus val(v)$)
XOR	$u = \mathbf{1}$		NOT(v)
XOR	$v = \mathbf{1}$		NOT(u)
XOR	$u = \mathbf{0}$	S	v
XOR	$v = \mathbf{0}$	S	u
XOR	$u = v$	S	CONST(0)
XOR	$u =$ NOT(v)	S, C	CONST(1)

S: Only with a shared representation
C: Only when complement edges are used

Fig. 5 Special cases for the Apply operation, with arguments *op*, u, and v

The other operations listed in Fig. 1 are implemented in a similar fashion. We give only brief descriptions here; more details can be found in [15]. The NOT operation proceeds by generating a copy of the argument OBDD, with the values of the leaf vertices inverted. To compute RESTRICT(f, i, b), we want to eliminate every vertex v in the graph for f having $var(v) = x_i$ and redirect each incoming arc to either $hi(v)$ (when $b = 1$) or $lo(v)$ (when $b = 0$). Rather than modifying existing vertices, we create new ones as needed, applying the reduction rules in the process.

We have already seen that the composition, quantification, and relational product operations can be computed using combinations of restriction and Boolean algebraic operations (Eqs. (2)–(4)). However, these operations are of such critical importance in symbolic model checking and other applications that they are often implemented with more specialized routines. As with Apply, these algorithms use a combination of depth-first traversal, memoizing, and the unique table to generate a reduced graph as their result.

As an example, the algorithm to perform existential quantification can be expressed as a recursive function EXISTS(u, I), where u is an OBDD vertex and I is a set of variable indices. It maintains a *quantifier cache* using keys of the form $\langle u, I \rangle$. On a given invocation, if neither a previously computed result is found nor a terminal case applies, it retrieves u_1 and u_0, the two children of u, and recursively computes $w_1 =$ EXISTS($u_1, I - \{i\}$) and $w_0 =$ EXISTS($u_0, I - \{i\}$). For $x_i = var(u)$, when $i \in I$, it computes the result as $w =$ APPLY(OR, u_1, u_0). Otherwise, it either

retrieves or creates a vertex w with $var(w) = x_i$, $hi(w) = w_1$, and $lo(w) = w_0$. Vertex w is then added to the quantifier cache with key $\langle u, I \rangle$. Universal quantification can be implemented similarly, or we can simply make use of De Morgan's Laws to express universal quantification in terms of existential: $\forall X.f = \neg\exists X.(\neg f)$.

As mentioned earlier, the relational product operation implements $\exists X.(f \wedge g)$ for variables X, and functions f and g. In principle, this operation could proceed by first computing $f \wedge g$ and then existentially quantifying the variables in X. Experience has shown, however, that the graph representing $f \wedge g$ will often be of unmanageable size, even though the final result of the relational product is more tractable. By combining conjunction and quantification during a single traversal of the graphs for f and g, this problem of "intermediate explosion" can often be avoided. This algorithm is expressed by a recursive function RELPROD(f, g, I) that uses a combination of the rules we have seen in the implementations of APPLY and EXISTS [20, 67].

We are left with the operations that test or examine one or more functions. As already mentioned, when using a shared representation, testing for equality can be done by simply checking whether the argument vertices are the same. With a split representation, we can implement a simple traversal of the two graphs to test for isomorphism. To evaluate a function for a specified set of argument values, we follow a path from the root to a leaf, at each step branching according to the value associated with the variable, with the leaf value serving as the result of the evaluation.

To find a single satisfying assignment for a function, we can search for a path from the root to the 1-leaf. This search does not require any backtracking, since, with the exception of arcs leading directly to the 0-leaf, each arc is part of a path leading to the 1-leaf. To find all satisfying solutions, we can perform a depth-first traversal of the graph to enumerate every path leading to the 1-leaf.

7.6 Implementation Techniques

Dozens of OBDD software packages have been created, displaying a variety of implementation strategies and features. Most implementations follow a set of principles described in a 1990 paper by Brace, Rudell, and Bryant (BRB) [12]. In an evaluation of many different packages for benchmarks arising from symbolic model-checking problems [67], the best performance consistently came from packages very similar to the BRB package. Here we highlight some of its key features. In Sect. 7.10, we describe efforts to scale OBDD implementations to handle large graphs and to support parallel execution. An excellent discussion of implementation issues can be found in [63].

Most OBDD packages, including BRB, use a shared representation, with all functions represented by a single, multi-rooted graph [52]. Formally, we can define a shared OBDD as representing a set of functions $\mathcal{F}$, where each $f \in \mathcal{F}$ is designated by a root vertex in the graph. As we have seen, this approach has several advantages over a separate representation:

- it reduces the total number of vertices required to represent a set of functions,
- it simplifies the task of checking for equality, and
- it provides additional cases where the recursions for operations such as APPLYand RESTRICT can be terminated (see Fig. 5)

On the other hand, using a shared representation introduces the need for some form of garbage collection to avoid having the available space exhausted by vertices that are no longer reachable from any of the active root vertices. Most shared OBDD implementations maintain a count of the total number of references to each vertex, including arcs from other vertices as well as external references to the root vertices. A vertex is a candidate for reclamation when its reference count drops to zero. Reclaiming a vertex also involves removing the corresponding entry from the unique table as well as every entry in the computed cache that references that vertex as part of its key or value.

The BRB package makes use of *complement edges*, where each edge has an additional attribute indicating whether or not the designated function is represented in true or complemented form. By adopting a set of conventions on the use of these attributes, it is possible to define a canonical form such that the NOT operation can be computed in constant time by simply inverting the attribute at the root [12, 45, 52]. By sharing the subgraphs for functions and their complements, such a representation can reduce the total number of vertices by as much as a factor of two. Perhaps more importantly, it makes it possible to perform the NOT operation in unit time. We can also see from Fig. 5 that the combination of a shared representation and complement edges provides additional terminal cases for Apply and other operations.

The BRB package generalizes the two-operand Boolean operations to a single three-argument operation known as *ITE* (short for "If-Then-Else"), defined as:

$$ITE(f, g, h) = (f \wedge g) \vee (\neg f \wedge h). \tag{7}$$

Using this single operation, we can implement other operations as:

$$\begin{aligned}
\text{AND}(f, g) &= ITE(f, g, \mathbf{0}) \\
\text{OR}(f, g) &= ITE(f, \mathbf{1}, g) \\
\text{XOR}(f, g) &= ITE\big(f, \text{NOT}(g), g\big) \\
\text{COMPOSE}(f, i, g) &= ITE\big(g, \text{RESTRICT}(f, i, 1), \text{RESTRICT}(f, i, 0)\big).
\end{aligned}$$

By rearranging and complementing the arguments according to a simple set of transformations, unifying the algebraic operations in this form can take advantage of De Morgan's Laws to increase the hit rate of the computed cache [12]. This can dramatically improve overall performance, since each hit in the computed cache can potentially eliminate many recursive calls.

One feature of BRB and most other packages is that the individual node data structures are *immutable*. During program execution, new nodes are created, and ones that are no longer needed can be recycled via garbage collection, but the nodes

themselves are not altered.[1] This functional programming model provides a useful abstraction for a Boolean function API, but it also implies that the package can expend much of its effort performing memory management tasks. New nodes must be created and old ones recycled, rather than simply letting the program modify existing nodes.

Several OBDD packages have been implemented that instead view the OBDD as a mutable data structure. For example, the SMART model checker [24] represents the set of states that have been encountered during a state-space exploration using a variant of OBDDs, called multi-valued decision diagrams (MDDs), that we describe in Sect. 7.9. As new states are encountered, the MDD is modified directly to include these states in its encoding. When performing model checking of asynchronous systems, as is the case with SMART, this approach seems appropriate, since each action of the system can be captured by a small change to the MDD.

7.7 Variable Ordering and Reordering

The algorithms we have presented require that the variables along all paths for all represented functions follow a common ordering. Any variable ordering can be used, and so the question arises: "How should the variable ordering be chosen?" Some functions are very sensitive to variable ordering, ranging from linear to exponential in the number of variables. These include the functions for bit-level representations of integer addition and comparison. Others, including all symmetric functions, remain of polynomial size for all variable orderings [15]. Still others have exponential size for all possible variable orderings, including those for a bit-level representation of integer multiplication [16].

We can express the choice of variable ordering by considering the effect of permuting the variables in the OBDD representation of a function. That is, for Boolean function f and permutation π over $\{1, \ldots, n\}$, define $\pi(f)$ to be a function such that

$$\pi(f)(x_1, \ldots, x_n) = f(x_{\pi(1)}, \ldots, x_{\pi(n)}).$$

Different permutations π yield different OBDDs, but all of these can be viewed as just different representations of a single underlying function. The task of finding a good variable ordering for a function f can then be defined as one of finding a permutation π that minimizes the number of vertices in the OBDD representation of $\pi(f)$. For a shared OBDD representation, we wish to find a good variable ordering for the entire graph. That is, for permutation π and function set $\mathcal{F}$, define $\pi(\mathcal{F})$ to be $\{\pi(f) \mid f \in \mathcal{F}\}$. For a shared OBDD implementation, we seek a permutation π that minimizes the number of vertices in the OBDD representation of $\pi(\mathcal{F})$.

[1] There is a nuance to this statement that we will discuss when we consider the implementation of dynamic variable reordering.

In general, the task of finding an optimal ordering π for a function f is NP-hard, even when f is given as an OBDD [9]. There is not even a polynomial-time algorithm that can guarantee finding a variable ordering within a constant factor of the optimum, unless $P = NP$ [61]. Similar results hold for a shared OBDD representation [65]. Published algorithms to find the exact optimal ordering have worst-case time complexity $O(n3^n)$ [32] for a function with n variables. Knuth has devised clever data structures that make the process practical for up to around $n = 25$ [41].

Instead of attempting to find the best possible ordering, a number of researchers have derived heuristic methods that have been found to generate reasonably good variable orders for specialized applications, such as when the Boolean function is derived from a combinational circuit [33, 46], a sequential circuit [39], a CNF representation [3], or a set of interacting state machines [6].

An alternate approach to finding a good variable ordering at the outset of the computation is to dynamically reorder the variables as the BDD operations proceed. This idea was introduced by Rudell [59], based on the observation that exchanging two adjacent variables in a shared OBDD representation can be implemented without making major changes to the Boolean function library API. Let π_i be the permutation that exchanges the values of i and $i + 1$, while keeping all other elements the same. Exchanging variables i and $i + 1$ in a shared OBDD representation involves converting the OBDD representation of function set $\mathcal{F}$ into one for function set $\pi_i(\mathcal{F})$. This transformation can be implemented by introducing new vertices and relabeling and eliminating some of the existing vertices, but with the property that the identities of all root vertices are preserved. This is an important property, since external references to functions being manipulated by the application program consist of pointers to root vertices in the graph. Thus, the reordering can be performed without altering any of these external references. Even though the relabeling of vertices mutates the node data structures, these changes still preserve the "functional" property stated earlier—the underlying functions being represented do not change.

Using pairwise exchanges of adjacent variables as the basic operation, most OBDD libraries implement dynamic variable ordering by a process known as *sifting* [59]. A single variable, or a small set of variables [57], is moved up and down in the ordering via a sequence of adjacent-variable exchanges, until a location yielding an acceptable number of total vertices is identified. In the original formulation of sifting, a variable is moved across the entire range of possible positions and then back to the position that minimizes the overall OBDD size. More recently, lower bound techniques have been used to guide the range over which each variable is moved [30]. Sifting is a very time-consuming process, but it has been shown to greatly improve memory performance—often the limiting factor for OBDD-based applications.

7.8 Variant Representations

Researchers have examined many variants of OBDDs, both for representing Boolean functions and for extending to functions where the domain, the range, or

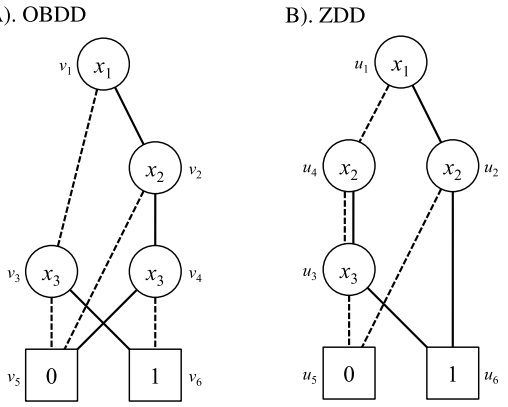


Fig. 6 BDD and ZDD representations of the set of sets {{1, 2}, {3}, {2, 3}}

both are non-Boolean. Here we survey some of the variants that have either proved effective for model checking or that seem especially promising. Some of these were also described in an earlier survey [18]. Other surveys provide even more comprehensive coverage of the many innovative variants of OBDDs that have been devised [31].

Zero-Suppressed BDDs

Perhaps the most successful variant of OBDDs are *zero-suppressed* BDDs [51], sometimes referred to as ZDDs. This representation differs from traditional OBDDs only in the interpretation applied to the case where an arc skips one or more variables. That is, it concerns the case where there is an arc emanating from a vertex v with label $var(v) = x_i$ to a vertex u with label $var(u) = x_j$, such that $j > i + 1$. In the example of Fig. 2 (reproduced on the left-hand side of Fig. 6), such an arc occurs from the root vertex v_1 to vertex v_3. With conventional OBDDs, such an arc indicates a case where the represented function is independent of any of the intervening variables. In the example, $f|_{x_1 \leftarrow 0}$ is independent of x_2. With a ZDD, such an arc indicates a case where the represented function is of the form $\neg x_{i+1} \wedge \cdots \wedge \neg x_{j-1} \wedge f_u$, where f_u is the function associated with vertex u. For ZDDs, we replace the second reduction rule for OBDDs (that a vertex cannot have two identical children) with a rule that no vertex can have the 0-leaf as its *hi* child.

More formally, we can define the Boolean function denoted by a ZDD by defining a set of functions of the form f_v^j for each vertex v. For leaf vertex v, we define this set for all $j \leq n + 1$ as follows:

$$f_v^j = \begin{cases} \mathbf{1}, & j = n+1 \text{ and } val(v) = 1 \\ \mathbf{0}, & j = n+1 \text{ and } val(v) = 0 \\ \neg x_j \wedge f_v^{j+1}, & j \leq n. \end{cases}$$

For nonterminal vertex v having $x_i = var(v)$ we define this set for all $j \leq i$ as:

$$f_v^j = \begin{cases} x_i \wedge f_{hi(v)}^{i+1} \vee \neg x_i \wedge f_{lo(v)}^{i+1}, & j = i \\ \neg x_j \wedge f_v^{j+1}, & j < i. \end{cases}$$

The function associated with root vertex v is then f_v^1.

Although ZDDs can be considered an alternate representation for Boolean functions, it is more useful to think of them as representing sets of sets. That is, let $M_n = \{1, \ldots, n\}$, and consider sets of sets of the form $S \subseteq \mathcal{P}(M_n)$. We can encode any set $A \subseteq M_n$ with a Boolean vector $\mathbf{a}$, where a_i equals 1 when $i \in A$, and equals 0 otherwise. The set of sets represented by Boolean function f consists of those sets A for which the corresponding Boolean vector yields $f(\mathbf{a}) = 1$. As examples, Fig. 6 shows both the OBDD (left) and the ZDD (right) representations of the set of sets $\{\{1, 2\}, \{3\}, \{2, 3\}\}$. The OBDD representation is identical to that of Fig. 2, because these are the only three satisfying assignments to Eq. (5). Comparing the ZDD, we see that, with two exceptions, each vertex v_i in the OBDD has a direct counterpart u_i in the ZDD. The first exception is the introduction of new vertex u_4 having two identical children, since such vertices are no longer eliminated by our revised reduction rules. The second exception is that there is no counterpart to vertex v_4, since this vertex had the 0-leaf as its *hi* child.

ZDDs are especially well suited for representing sets of sparse sets, defined as having two general properties:

- The total number of sets is much smaller than 2^n.
- Most of the included sets have far fewer than n elements.

These conditions tend to give OBDD representations where many nonterminal vertices have the 0-leaf as their *hi* children, and these vertices are eliminated by using a ZDD representation.

The OBDD and ZDD representations of a function do not differ greatly in size. It can easily be shown that if these two representations have N_o and N_z vertices, respectively, then $N_z/n \leq N_o \leq n \times N_z$ [60]. Nonetheless, for complex functions and large values of n, the advantage of one representation over the other can be very significant.

ZDDs have proved especially effective for encoding combinatorial problems [41, 60]. They have been used in model checking for cases where the set of states has the sparseness properties we have listed, such as for Petri nets [69].

Partitioned OBDDs

The general principle of partitioned OBDDs is to divide the 2^n possible combinations of variable assignments into m different, nonoverlapping subsets, and then create a separate representation for a function over each subset.

More formally, define a set of *partitioning functions* as a set of functions $P = \langle p_1, \ldots, p_m \rangle$, such that $\bigvee_i p_i = \mathbf{1}$ and for each i and j such that $i \neq j$, we have $p_i \wedge p_j = \mathbf{0}$.

Each function f is then represented by a set of functions $\langle f_1, \ldots f_m \rangle$, where each f_i equals $f \wedge p_i$. It can readily be seen that the Boolean operations distribute over any partitioning. For example, for $h = f \vee g$, we have $h_i = f_i \vee g_i$ for each partition i. On the other hand, other operations, including restriction, quantification, and composition do not, in general, distribute over a partitioning.

Partitioning has been shown to be effective for applications where conventional, monolithic OBDDs would be too large to represent and manipulate. One approach is to allow different variable orderings for each partition [54]. This approach works well for applications where some small set of "control" variables determine important properties of how the remaining variables relate to one another. The different partitions then consist of all possible enumerations of these control variables.

As will be discussed later (Sect. 7.10), partitioning can also provide the basis for mapping an OBDD-based application onto multiple machines in a distributed computing environment.

7.9 Representing Non-Boolean Functions

Many systems for which we might wish to apply model checking involve state variables or parameters that are not Boolean. A number of schemes have been devised to represent such functions as decision diagrams, seeking to preserve the key properties of OBDDs: (1) they achieve compactness, mostly through the sharing of subgraphs, (2) key operations can be implemented via graph algorithms, and (3) properties of the represented functions can readily be tested. Here we describe some of the decision diagrams that have been used in model checking and related applications.

Functions over Discrete Domains

Consider the case where function variable x ranges over a finite set $D = \{d_0, \ldots, d_{K-1}\}$. There are several possible ways to represent a function over x as a decision diagram:

Binary encoding: Recode x in terms of Boolean variables $x_{k-1}, x_{k-2}, \ldots, x_0$, where $k = \lceil \log_2 K \rceil$. Each value d_i is encoded according to the binary representation of i. When K is not a power of 2, then we can either (1) add an additional constraint that any valid assignment to the Boolean variables must correspond to a binary value less than K, or (2) define multiple assignments to the Boolean variables to encode a single value from D. A binary encoding minimizes the number of Boolean variables required.

Unary encoding: Recode x in terms of Boolean variables $b_{K-1}, b_{K-2}, \ldots, b_0$, where value d_i is encoded by having $x_i = 1$ and all other values equal to zero. Except for very small values of K, this encoding would be impractical for OBDDs, but it works well for ZDDs.

Multiway branching: Generalize the OBDD data structure to *multi-valued* decision diagrams [24, 40], where a vertex for a K-valued variable has an outgoing arc for each of its K children.

Indeed, all three of these approaches have been used successfully.

For representing functions over discrete domains having non-Boolean ranges, the most straightforward approach is to allow the leaves to have arbitrary values, leading to *multi-terminal* binary decision diagrams (MTBDDs) [34]. (These have also been called *algebraic* decision diagrams (ADDs) [7].) More precisely, for a function f mapping to some codomain R, define its *image* $Img(f)$ as those values $r \in R$ such that $r = f(\mathbf{a})$ for some argument value $\mathbf{a}$. Then the MTBDD representation of f has a leaf vertex for each value in $Img(f)$.

The set of operations on such functions depends on the type of functions being represented. Typically, they follow the same approach we saw with the algorithm for the Apply operation (Sect. 7.5)—they recursively traverse the argument graphs, stopping when either a terminal case is reached, or the arguments match those stored in a computed cache. For example, when R is either the set of reals or integers, such an approach can be used to perform algebraic operations such as addition or multiplication over functions. It can also be used to generate a *predicate*, capturing some property of the function values. For example, for function f mapping to real values, let Z_f be the Boolean function that yields 1 for those arguments $\mathbf{a}$ for which $f(\mathbf{a}) = 0.0$, and 0 otherwise. We can generate an OBDD representation of Z_f by traversing the MTBDD representation of f, returning $\mathbf{1}$ when we encounter leaf value 0.0, $\mathbf{0}$ when we encounter a nonzero leaf value, and either generating a new vertex or retrieving one from the unique table for the nonterminal cases.

MTBDDs have been used for a variety of applications, encoding such values as data-dependent delays in transistor circuits [48], as well as transition probabilities in Markov chains [43]. Their biggest limitation is that the size of a function image can be quite large, possibly exponential in the number of function variables. Such a function will have many leaf vertices and therefore little sharing of subgraphs. This lack of sharing will reduce the advantage of decision diagrams over more direct encodings of the problem domain, both in the compactness of the representation and the speed of the operations on them. Successful applications of MTBDDs often avoid this "value explosion" by exploiting the modularity in the underlying system. For example, when performing model checking of stochastic systems, the transition probabilities for the different subsystems can be maintained as separate MTBDDs, rather than combined via a product construction [2].

Functions over Unbounded Domains

When a function variable x ranges over an infinite domain D, we cannot simply encode its possible values with a set of binary values or add multiple branches to the vertices of a decision diagram. In some applications, however, we need only capture a bounded set of attributes of the state variables. In this section, we describe

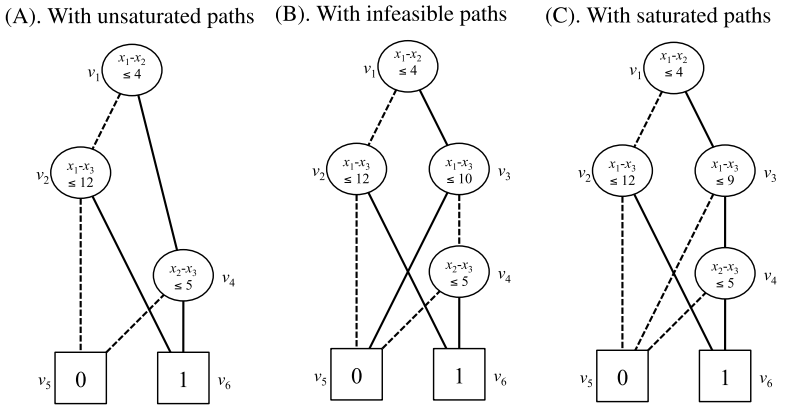


Fig. 7 Difference Decision Diagram (DDD) Examples. Vertices are labeled by difference constraints

Difference Decision Diagrams (DDDs) [53] as an example of this approach. DDDs illustrate a general class of decision diagrams, where the decisions are based on predicates over some domain, rather than simple Boolean variables. We then discuss several variants and extensions of this representation.

The *Difference Decision Diagram* data structure was devised specifically for analyzing timed automata. As discussed in Chap. 29 [11], a timed automaton operates over a discrete state space but also contains real-valued clocks that all proceed at the same rate, but can be at different offsets with respect to one another [4]. Although the clock values can be unbounded, their behavior can be characterized during model checking in terms of a finite set of bounds on their differences. DDDs therefore express the values of the clocks in terms of a set of *difference constraints*, each of the form $x_i - x_j \leq c$ or $x_i - x_j < c$, where x_i and x_j are clock variables, and c is an integer or real value. In the spirit of OBDDs, DDDs also impose an ordering requirement over difference constraints, based on the indices i and j of the two variables, the comparison operator ($\leq$ vs. $<$), and the constant c.

Figure 7 show three examples of DDDs and serves to illustrate some subtle issues that arise when generalizing from a decision diagram where the decisions represent independent Boolean variables to one in which the decisions represent predicates over some other domain. The DDD on the left (A) represents a disjunction of two different constraints C_1 and C_2, defined as follows:

$$C_1 = (x_1 - x_2 > 4) \wedge (x_1 - x_3 \leq 12)$$
$$C_2 = (x_1 - x_2 \leq 4) \wedge (x_2 - x_3 \leq 5)$$

The DDD in the center (B) also represents a disjunction of two constraints: C_1, as in (A), as well as a constraint C_2':

$$C_2' = (x_1 - x_2 \leq 4) \wedge (x_2 - x_3 \leq 5) \wedge (x_1 - x_3 > 10).$$

On closer examination, however, we can see that constraint C_2' must be false for all values of x_1, x_2, and x_3. That is, if $x_1 - x_2 \leq 4$ and $x_2 - x_3 \leq 5$, then we must have $x_1 - x_3 \leq 9$, and this conflicts with the term $x_1 - x_3 > 10$. The possibility of infeasible paths implies that there is no simple way to determine whether a set of constraints represented as a DDD is satisfiable, whereas this is a trivial task with OBDDs. In particular, it is possible to determine whether any path in a DDD from the root to the 1-leaf is satisfiable in polynomial time, but there can be an exponential number of such paths.

The DDD on the right (C) represents a disjunction of constraint C_1, as before, and a constraint C_2'':

$$C_2'' = (x_1 - x_2 \leq 4) \wedge (x_2 - x_3 \leq 5) \wedge (x_1 - x_3 \leq 9).$$

We can see that constraint C_2'' is mathematically equivalent to C_2; the first and second terms of C_2'' already imply that the third term, $x_1 - x_3 \leq 9$, is redundant. In fact, constraint C_2'' is *saturated*, meaning that it contains a predicate for every pairwise constraint that can be inferred from it.

These examples show how the interdependencies between the predicates can lead to paths in a DDD that are infeasible, as well as ones where different combinations of terms can be mathematically equivalent. The developers of DDDs describe an algorithm that eliminates infeasible paths by testing each one individually and restructuring the DDD when an infeasible path is found [53]. In the worst case, this process can require time exponential in the size of the DDD, and it can also increase its size. Once infeasible paths have been eliminated, then satisfiability becomes easily testable. The developers also propose several rules for dealing with redundant tests, including ensuring that every path is saturated. This leads to a form that they conjecture is canonical, although this has apparently never been proven. Fortunately, most of the algorithms that use DDDs do not require having a canonical representation.

Several other decision diagrams have been devised specifically for model checking of timed automata. Clock difference diagrams [44] coalesce the predicates of difference decision diagrams, such that along any path there is a single node representing all constraints on a given pair of variables x_i and x_j. This node has multiple outgoing branches, corresponding to disjoint intervals representing possible values for $x_i - x_j$. Clock restriction diagrams [66] also have multiple branches emanating from a single node associated with variables x_i and x_j, but these represent possible upper bounds on the value of $x_i - x_j$. (Lower bounds on this value are represented as upper bounds on the value of $x_j - x_i$.)

Although the focus of much of the work in representing constraints on real-valued variables was motivated by the desire to perform symbolic model checking on timed automata, such constraints arise in other applications as well. DDDs can represent difference constraints of the form $x_i - x_j \leq c$. Other constraints of interest include box constraints of the form $x_i \leq c$, or more generally, arbitrary linear constraints of the form $a_1 \cdot x_1 + a_2 \cdot x_2 + \cdots + a_n \cdot x_n \leq c$. Linear decision diagrams extend DDDs to include such constraints [23]. With both DDDs and LDDs, it is also possible to have nodes labeled by Boolean variables as well as ones labeled

by constraints. Such decision diagrams can be used when verifying hybrid systems, containing both continuous and discrete state variables.

We can see a parallel between these different forms of decision diagrams and SMT solvers (Chap. 11 [8]). Just as SMT extends Boolean satisfiability solvers to implement decision procedures for other mathematical theories, these generalizations of decision diagrams extend OBDDs to symbolically represent functions over other theories. Both must deal with cases where some combination of constraints is infeasible, leading to conflicts in SMT solvers and infeasible paths in decision diagrams.

7.10 Scaling OBDD Capacity

Although the introduction of OBDD-based symbolic model checking in the early 1990s provided a major breakthrough in the size and complexity of systems that could be verified, the nature of our field and our desire to apply our tools to real-world systems means that we will always seek to scale them to handle ever larger and more complex problems. Computer systems continue to scale—individual machines have more memory, more cores, and larger disk capacity. In addition, we routinely map problems onto larger clusters of machines that are programmed to work together on a single task. One would expect BDD libraries to have evolved to take advantage of these technological advantages, but unfortunately this is not the case. Most widely used BDD packages still execute on a single core of a single machine, and they are barely able to use the amount of physical memory available on high-end machines. In this section, we highlight some of the efforts to scale the capacity of OBDD implementations, and some of the challenges these efforts face.

In most applications of OBDDs, the ability to handle larger and more complex problems is limited more by the size of the OBDDs generated, rather than the CPU performance. In the extreme case, very large OBDDs can grow to exceed the memory capacity of a machine. On a 64-bit machine, storing the OBDD nodes and all of the associated tables requires, on average, around 40 bytes per node. Thus a machine with 16 GB of RAM should, in principle, be able to support OBDD applications using up to around 400 million nodes. In practice, however, the performance of most OBDD implementations becomes unacceptably slow well before that point, due to poor memory-system performance. Traversing graphs in depth-first order (as occurs with the recursive implementation of the Apply algorithm described in Sect. 7.5) tends to yield poor virtual memory and cache performance, due to a lack of locality in the memory access patterns. Standard implementations of the hash tables used for the unique table and the computed cache also exhibit poor memory locality.

Some efforts have been made to implement OBDDs with an eye toward memory performance [5, 55, 58, 68]. These typically employ breadth-first traversal techniques and try to pack the vertices for each level into a contiguous region of memory. A breadth-first approach also lends itself to an implementation where most of the data are stored on a large disk array [42]. Unfortunately, none of these ideas seem to have been incorporated into publicly available OBDD packages.

Early efforts to exploit parallelism in OBDD operations demonstrated the difficulty of this task. Most were implemented in a "shared nothing" environment, where each processor has its own independent memory and can only communicate with other processors via message passing. These implementations require some strategy for *partitioning* the OBDD, so that each node is assigned to some processor. In a message-passing environment, traversing a graph that is partitioned across machines requires message communications, versus the simply memory referencing that occurs on a single machine, and so the performance improvements due to greater parallelism must overcome the potentially high cost of node referencing. Implementations based on a random partitioning of the nodes [64] only showed performance superior to a sequential implementation when the size of the graph exceeded the capacity of a single processor's memory. In an attempt to minimize the need for message passing, other implementations used a layered partitioning, where the range of variable indices is divided into subranges, and all nodes within a given subrange are mapped onto a single machine. Implementations that were specialized to symbolic model checking could use a partitioning where different regions of the state space were mapped onto different machines [37], following the principles of partitioned OBDDs.

The recent availability of multicore processors supporting multiple threads executing within a single memory space has revived interest in exploiting parallelism in OBDD operations. There are two natural sources of parallelism: *internal*, in which individual operations such as Apply use multiple threads [29], and *external*, in which a multi-threaded application can invoke multiple Apply operations concurrently [56]. An implementation that uses only internal parallelism requires no changes to the API, while those that support external parallelism can use some mechanism, such as futures, to allow one thread to invoke an operation on OBDDs that are still being generated by other threads.

With the entire OBDD and all of the tables held in a shared memory, any core can access any node or table entry via a memory reference. Obtaining good performance requires careful attention to memory locality and to the potential for thrashing, where multiple threads compete to read and write a small number of cache lines. Such thrashing can occur due to poor design of user data structures or due to excessive calls to synchronization primitives. Excessive synchronization can also lead to a loss of parallelism among the threads.

Perhaps the most ambitious attempt to map an OBDD implementation onto multicore processors has been by researchers at the University of Twente [29]. Their system maintains a set of workers, each of which maintains a queue of tasks. The system implements the Apply operation with a task for each recursive step. To perform the recursion, each task then spawns two new tasks, with one performed by the current worker and the other added to the worker's queue. Workers are kept busy by having them execute the tasks in their own queues, and "stealing" tasks from other queues when needed. As the computation unfolds, this overall approach will have the effect of having many workers collaboratively executing the Apply operation over different parts of the argument graphs. The system maintains a single unique table and a single computed cache as a way of maintaining consistency and avoiding

duplicate efforts by the workers. By carefully designing these tables to use lockless synchronization and cache-friendly data structures, they are able to achieve high performance.

Building a multi-processor system with coherent shared memory becomes prohibitively expensive as the system scales to thousands of processors. Thus, an important challenge remains to devise OBDD implementations that can operate effectively in a fully distributed, shared-nothing environment.

Comparison to SAT Checking

We conclude with some observations about how OBDD-based reasoning systems and propositional satisfiability (SAT) checkers have important similarities and differences, both from conceptual and operational viewpoints. Clearly, both are related in the sense that they solve problems encoded in Boolean form. On the other hand, they differ greatly in their intended task—a SAT checker need only find a single satisfying assignment to a Boolean formula, while converting a Boolean formula to an OBDD creates an encoding that describes all of its satisfying solutions. Once we have generated the OBDD representation, it becomes straightforward to perform tasks that SAT solvers cannot readily do, such as counting the number of solutions, or finding an optimal solution for some cost function. Furthermore, OBDDs support operations, such as variable quantification, that have proved to be very challenging extensions for SAT checking.

For most applications of satisfiability testing, SAT checkers based on the Davis–Putnam–Logemann–Loveland (DPLL) algorithm [27, 28] (Chap. 9 [47]) greatly outperform ones that construct an OBDD and then call the SATISFY operation to generate a solution. There are some notable exceptions, however. For example, Bryant conducted experiments on satisfiability problems to test the equivalence of *parity trees*—networks of exclusive-or logic gates computing the odd parity of a set of n Boolean values [19]. Each experiment tested whether a randomly generated tree was functionally equivalent to one consisting of a linear chain of logic gates. We performed tests using four state-of-the-art SAT solvers, but none could handle cases of $n = 48$ inputs within a 900 second time limit. These parity tree problems are known to be difficult cases for DPLL, or in fact any method based on the resolution principle. By contrast, an OBDD-based solver could readily handle such problems in well under 0.1 seconds. Indeed, the OBDD representation of the parity function grows only linearly in n.

This example illustrates the opportunity to devise SAT checkers that combine top-down, search-based strategies, such as DPLL, with ones based on bottom-up, constructive approaches, such as OBDDs. One approach is to replace the traditional clause representation of SAT solvers with OBDDs, where the task becomes to find a single variable assignment that yields 1 for all of the OBDDs [26]. Beyond the usual steps of a SAT solver, the solver can also replace some subset of the OBDDs with their conjunction. This approach can deal with problems for which OBDDs

outperform DPLL (e.g., the parity tree example), while also getting the performance advantages of DPLL-based SAT solvers.

Other connections between OBDDs and DPLL-based SAT solvers arise due to the observation that the search tree generated by DPLL bears much resemblance to an OBDD: each selection of a decision variable in DPLL creates a vertex in the search tree, with outgoing branches based on the value assigned to the variable. Depending on the decision heuristic used, DPLL might follow a common variable ordering across the entire tree, yielding a tree that obeys the ordering constraint of OBDDs, or it may have different orderings along different paths. These are analogous to a class of BDDs known as "free BDDs," in which variables can occur in any order from the root to a leaf in the graph, but no variable can occur more than once [36].

Huang and Darwiche exploit this relationship to modify an existing DPLL-based SAT solver to instead generate the OBDD representation of a formula given in CNF form [38]. They found this top-down approach to OBDD construction fared better for formulas expressed in CNF than did the usual bottom-up method based on the Apply algorithm. Along related lines, methods have been developed to analyze the clausal representation of a formula and generate a variable ordering that should work well for either SAT checking or for OBDD construction [3].

7.11 Concluding Remarks

Symbolic model checking arose by linking a model checking algorithm based on fixed-point computations with binary decision diagrams to represent the underlying sets and transition relations [10, 21, 25, 49]. This yielded a major breakthrough in the size and complexity of systems that could be verified. Since that time, OBDDs have been applied to many other tasks, but model checking remains one of their most successful applications. Even as model checkers have been extended to use other reasoning methods, especially Boolean satisfiability solvers, OBDDs have still proved valuable for supporting the range of operations required to implement full-featured model checkers.

Several major goals drive continued research on OBDDs and related representations. First, the desire to represent larger functions requires scaling OBDD implementations to exploit the memory sizes and multicore capabilities of modern processors, as well as large-scale, cluster-based systems. Second, possible variants on OBDDs may enable them to represent Boolean functions in more compact forms. Finally, the desire to verify systems having state variables that range over larger discrete domains, as well as infinite domains, provides a motivation to create types of decision diagrams that can represent other classes of functions.

The resulting research efforts continue to yield novel ideas and approaches, while taking advantage of the key property of OBDDs: that they can represent a variety of functions in a compact form, and that they can be constructed and analyzed using efficient graph algorithms. Future developments will certainly enhance the ability of OBDD-based methods to support model checking.

References

1. Akers, S.B.: Binary decision diagrams. IEEE Trans. Comput. **C-27**(6), 509–516 (1978)
2. de Alfaro, L., Kwiatkowska, M., Parker, G.N.D., Segala, R.: Symbolic model checking of probabilistic processes using MTBDDs and the Kronecker representation. In: Graf, S., Schwartzbach, M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 1785, pp. 395–410. Springer, Heidelberg (2000)
3. Aloul, F.A., Markov, I.L., Sakallah, K.A.: Faster SAT and smaller BDDs via common function structure. In: Proc. of the Intl. Conf. on Computer-Aided Design (ICCAD), pp. 443–448. IEEE, Piscataway (2001)
4. Alur, R., Dill, D.: A theory of timed automata. Theor. Comput. Sci. **126**(2), 183–235 (1994)
5. Ashar, P., Cheong, M.: Efficient breadth-first manipulation of binary decision diagrams. In: Proc. of the Intl. Conf. on Computer-Aided Design (ICCAD), pp. 622–627. IEEE, Piscataway (1994)
6. Aziz, A., Taşiran, S., Brayton, R.K.: BDD variable ordering for interacting finite state machines. In: Proc. of the 31st ACM/IEEE Design Automation Conf. (DAC), pp. 283–288. ACM/IEEE, New York/Piscataway (1994)
7. Bahar, R.I., Frohm, E.A., Gaona, C.M., Hachtel, G.D., Macii, E., Pardo, A., Somenzi, F.: Algebraic decision diagrams and their applications. In: Proc. of the Intl. Conf. on Computer-Aided Design (ICCAD), pp. 188–191. IEEE, Piscataway (1993)
8. Barrett, C.W., Tinelli, C.: Satisfiability modulo theories. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
9. Bollig, B., Wegener, I.: Improving the variable ordering of OBDDs is NP-complete. IEEE Trans. Comput. **45**(9), 993–1002 (1996)
10. Bose, S., Fisher, A.L.: Automatic verification of synchronous circuits using symbolic logic simulation and temporal logic. In: Claesen, L. (ed.) Proc. of the IMEC-IFIP Intl. Workshop on Applied Formal Methods for Correct VLSI Design, pp. 759–764 (1989)
11. Bouyer, P., Fahrenberg, U., Larsen, K.G., Markey, N., Ouaknine, J., Worrell, J.: Model checking real-time systems. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
12. Brace, K.S., Rudell, R.L., Bryant, R.E.: Efficient implementation of a BDD package. In: Proc. of the 27th ACM/IEEE Design Automation Conf. (DAC), pp. 40–45. ACM/IEEE, New York/Piscataway (1990)
13. Brayton, R.K., Hachtel, G.D., McMullen, C.T., Sangiovanni-Vincentelli, A.L.: Logic Minimization Algorithms for VLSI Synthesis. Kluwer Academic, Norwell (1984)
14. Brown, F.M.: Boolean Reasoning. Kluwer Academic, Norwell (1990)
15. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Trans. Comput. **C-35**(8), 677–691 (1986)
16. Bryant, R.E.: On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication. IEEE Trans. Comput. **40**(2), 205–213 (1991)
17. Bryant, R.E.: Symbolic Boolean manipulation with ordered binary decision diagrams. ACM Comput. Surv. **24**(3), 293–318 (1992)
18. Bryant, R.E.: Binary decision diagrams and beyond: Enabling technologies for formal verification. In: Proc. of the Intl. Conf. on Computer-Aided Design (ICCAD), pp. 236–243. IEEE, Piscataway (1995)
19. Bryant, R.E.: A view from the engine room: Computational support for symbolic model checking. In: Grumberg, O., Veith, H. (eds.) 25 Years of Model Checking. LNCS, vol. 5000, pp. 145–149. Springer, Heidelberg (2008)
20. Burch, J.R., Clarke, E.M., Long, D.E., McMillan, K.L.: Symbolic model checking for sequential circuit verification. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. **13**(4), 401–424 (1994)
21. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. Inf. Comput. **98**(2), 142–170 (1992)

22. Chaki, S., Gurfinkel, A.: BDD-based symbolic model checking. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
23. Chaki, S., Gurfinkel, A., Strichman, O.: Decision diagrams for linear arithmetic. In: Biere, A., Pixley, C. (eds.) Formal Methods in Computer-Aided Design (FMCAD), pp. 53–60. IEEE, Piscataway (2009)
24. Ciardo, G., Marmorstein, R., Siminiceanu, R.: The saturation algorithm for symbolic state-space exploration. Int. J. Softw. Tools Technol. Transf. **8**, 4–25 (2006)
25. Coudert, O., Berthet, C., Madre, J.C.: Verification of synchronous sequential machines based on symbolic execution. In: Sifakis, J. (ed.) Proc. of the Workshop on Automatic Verification Methods for Finite State Systems. LNCS, vol. 407, pp. 365–373. Springer, Heidelberg (1989)
26. Damiano, R., Kukula, J.: Checking satisfiability of a conjunction of BDDs. In: Proc. of the 40th ACM/IEEE Design Automation Conf. (DAC), pp. 818–923. ACM/IEEE, New York/Piscataway (2003)
27. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Commun. ACM **5**(7), 394–397 (1962)
28. Davis, M., Putnam, H.: A computing procedure for quantification theory. J. ACM **3**, 201–215 (1960)
29. van Dijk, T., Laarman, A.W., van de Pol, J.C.: Multi-core BDD operations for symbolic reachability. In: 11th Intl. Workshop on Parallel and Distributed Methods in Verification (PDMC) (2012)
30. Drechsler, R., Günther, W., Somenzi, F.: Using lower bounds during dynamic BDD minimization. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. **20**(1), 51–57 (2001)
31. Drechsler, R., Sieling, D.: Binary decision diagrams in theory and practice. Int. J. Softw. Tools Technol. Transf. **3**(2), 112–136 (2001)
32. Friedman, S.J., Supowit, K.J.: Finding the optimum variable ordering for binary decision diagrams. IEEE Trans. Comput. **39**(5), 710–713 (1990)
33. Fujita, M., Fujisawa, H., Kawato, N.: Evaluation and improvements of Boolean comparison method based on binary decision diagrams. In: Proc. of the Intl. Conf. on Computer-Aided Design (ICCAD), pp. 2–5. IEEE, Piscataway (1988)
34. Fujita, M., McGeer, P.C., Yang, J.C.: Multi-terminal binary decision diagrams: an efficient data structure for matrix representation. Form. Methods Syst. Des. **10**, 149–169 (1997)
35. Garey, M.R., Johnson, D.S.: Computers and Intractability. Freeman, New York (1979)
36. Gunther, W., Drechsler, R.: Minimization of free BDDs. In: Proc. of the 1999 Conf. on Asia South Pacific Design Automation (ASP-DAC), pp. 323–326. IEEE, Piscataway (1999)
37. Heyman, T., Geist, D., Grumberg, O., Shuster, A.: Achieving scalability in parallel reachability analysis of very large circuits. In: Emerson, E.A., Sistla, A.P. (eds.) Proc. of the 12th Intl. Conf. of Computer Aided Verification (CAV). LNCS, vol. 1855, pp. 20–35. Springer, Heidelberg (2000)
38. Huang, J., Darwiche, A.: Using DPLL for efficient OBDD construction. In: Hoos, H.H., Mitchell, D.G. (eds.) Theory and Applications of Satisfiability Testing (SAT). LNCS, vol. 3542, pp. 157–172. Springer, Heidelberg (2005)
39. Jeong, S.W., Plessier, B., Hachtel, G.D., Somenzi, F.: Variable ordering and selection of FSM traversal. In: Proc. of the Intl. Conf. on Computer-Aided Design (ICCAD). IEEE, Piscataway (1991)
40. Kam, T., Villa, T., Brayton, R.K., Sangiovanni-Vincentelli, A.L.: Multi-valued decision diagrams: theory and applications. Mult. Valued Log. **4**(1–2), 9–62 (1998)
41. Knuth, D.S.: The Art of Computer Programming, vol. 4: Combinatorial Algorithms. Addison-Wesley, Reading (2011)
42. Kunkle, D., Slavici, V., Cooperman, G.: Parallel disk-based computation for large, monolithic binary decision diagrams. In: Maza, M.M., Roch, J.-L. (eds.) Proc. of the 4th Intl. Workshop on Parallel and Symbolic Computation (PASCO), pp. 63–72. ACM, New York (2010)
43. Kwiatkowska, M., Norman, G., Parker, D.P.: Probabilistic symbolic model checker. In: Field, T., Harrison, P.G., Bradley, J., Harder, U. (eds.) Computer Performance Evaluation: Modelling Techniques and Tools (TOOLS). LNCS, vol. 2324, pp. 113–140. Springer, Heidelberg (2002)

44. Larsen, K.G., Pearson, J., Weise, C., Yi, W.: Clock difference diagrams. Nord. J. Comput. **6**(3), 271–298 (1999)
45. Madre, J.C., Billon, J.P.: Proving circuit correctness using formal comparison between expected and extracted behaviour. In: Proc. of the 25th ACM/IEEE Design Automation Conf. (DAC), pp. 205–210. ACM/IEEE, New York/Piscataway (1988)
46. Malik, S., Wang, A., Brayton, R.K., Sangiovanni-Vincentelli, A.L.: Logic verification using binary decision diagrams in a logic synthesis environment. In: Proc. of the Intl. Conf. on Computer-Aided Design (ICCAD), pp. 6–9. IEEE, Piscataway (1988)
47. Marques-Silva, J., Malik, S.: Propositional SAT solving. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
48. McDonald, C.B., Bryant, R.E.: CMOS circuit verification with symbolic switch-level timing simulation. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. **20**(3), 458–474 (2001)
49. McMillan, K.L.: Symbolic Model Checking. Kluwer Academic, Norwell (1993)
50. Michie, D.: "Memo" functions and machine learning. Nature **218**, 19–22 (1968)
51. Minato, S.: Zero-suppressed BDDs for set manipulation in combinatorial problems. In: Proc. of the 30th ACM/IEEE Design Automation Conf. (DAC), pp. 272–277. ACM/IEEE, New York/Piscataway (1993)
52. Minato, S., Ishiura, N., Yajima, S.: Shared binary decision diagrams with attributed edges for efficient Boolean function manipulation. In: Proc. of the 27th ACM/IEEE Design Automation Conf. (DAC), pp. 52–57. ACM/IEEE, New York/Piscataway (1990)
53. Møller, J., Lichtenberg, J., Andersen, H., Hulgaard, H.: Difference decision diagrams. In: Flum, J., Rodriguez-Artalejo, M. (eds.) Computer Science Logic (CSL). LNCS, vol. 1683, p. 826. Springer, Heidelberg (1999)
54. Narayan, A., Jain, J., Fujita, M., Sangiovanni-Vincentelli, A.L.: Partitioned OBDDs—a compact, canonical, and efficiently manipulable representation for Boolean functions. In: Proc. of the Intl. Conf. on Computer-Aided Design (ICCAD), pp. 547–554. IEEE, Piscataway (1996)
55. Ochi, H., Yasuoka, K., Yajima, S.: Breadth-first manipulation of very large binary-decision diagrams. In: Proc. of the Intl. Conf. on Computer-Aided Design (ICCAD), pp. 48–55. IEEE, Piscataway (1993)
56. Ossowski, J.: JINC—a multi-threaded library for higher-order weighted decision diagram manipulation. Ph.D. thesis, Rheinische Friedrich-Wilhelms-Universität, Bonn (2009)
57. Panda, S., Somenzi, F.: Who are the variables in your neighbourhood. In: Formal Methods in Computer-Aided Design (FMCAD), pp. 74–77. IEEE, Piscataway (1995)
58. Ranjan, R.K., Sanghavi, J.V., Brayton, R.K., Sangiovanni-Vincentelli, A.L.: High performance BDD package based on exploiting memory hierarchy. In: Proc. of the 33rd ACM/IEEE Design Automation Conf. (DAC), pp. 635–640. ACM/IEEE, New York/Piscataway (1996)
59. Rudell, R.L.: Dynamic variable ordering for ordered binary decision diagrams. In: Proc. of the Intl. Conf. on Computer-Aided Design (ICCAD), pp. 139–144. IEEE, Piscataway (1993)
60. Schröer, O., Wegener, I.: The theory of zero-suppressed BDDs and the number of knight's tours. Form. Methods Syst. Des. **13**(3), 235–253 (1998)
61. Sieling, D.: On the existence of polynomial time approximation schemes for OBDD minimization. In: Morvan, M., Meinel, C., Krob, D. (eds.) Symp. on Theoretical Aspects of Computer Science (STACS). LNCS, vol. 1373, pp. 205–215. Springer, Heidelberg (1998)
62. Sieling, D., Wegener, I.: Reduction of OBDDs in linear time. Inf. Process. Lett. **48**(3), 139–144 (1993)
63. Somenzi, F.: Efficient manipulation of decision diagrams. Int. J. Softw. Tools Technol. Transf. **3**(2), 171–181 (2001)
64. Stornetta, T., Brewer, F.: Implementation of an efficient parallel BDD package. In: Proc. of the 33rd ACM/IEEE Design Automation Conf. (DAC), pp. 641–644. ACM/IEEE, New York/Piscataway (1996)
65. Tani, S., Hamaguchi, K., Yajima, S.: The complexity of the optimal variable ordering problems of shared binary decision diagrams. Algorithms Comput. **762**, 389–398 (1993)
66. Wang, F.: Efficient verification of timed automata with efficient BDD-like data structures. Int. J. Softw. Tools Technol. Transf. **6**(1), 77–97 (2004)

67. Yang, B., Bryant, R.E., O'Hallaron, D.R., Biere, A., Coudert, O., Janssen, G., Ranjan, R.K., Somenzi, F.: A performance study of BDD-based model checking. In: Gopalakrishnan, G., Windley, P. (eds.) Formal Methods in Computer-Aided Design (FMCAD). LNCS, vol. 1522. Springer, Heidelberg (1998)
68. Yang, B., Chen, Y.A., Bryant, R.E., O'Hallaron, D.R.: Space- and time-efficient BDD construction via working set control. In: Proc. of the 1998 Conf. on Asia South Pacific Design Automation (ASP-DAC), pp. 423–432. IEEE, Piscataway (1998)
69. Yoneda, T., Hatori, H., Takahara, A., Minato, S.: BDDs vs. zero-suppressed BDDs for CTL symbolic model checking of Petri nets. In: Srivas, M., Camilleri, A. (eds.) Formal Methods in Computer-Aided Design (FMCAD). LNCS, vol. 1166, pp. 435–449. Springer, Heidelberg (1996)

Chapter 8
BDD-Based Symbolic Model Checking

Sagar Chaki and Arie Gurfinkel

Abstract Symbolic model checking based on Binary Decision Diagrams (BDDs) is one of the most celebrated breakthroughs in the area of formal verification. It was originally proposed in the context of hardware model checking, and advanced the state of the art in model-checking capability by several orders of magnitude in terms of the sizes of state spaces that could be explored successfully. More recently, it has been extended to the domain of software verification as well, and several BDD-based model checkers for Boolean programs and push-down systems have been developed. In this chapter, we summarize some of the key concepts and techniques that have emerged in this story of successful practical verification.

8.1 Introduction

Algorithms for temporal logic model checking [14] were initially implemented in an explicit-state manner. This means that all automata involved in verification were represented using explicit graph-based data structures. Such automata include the Kripke structures as well as Büchi automata and tableaux obtained from the temporal logic specifications. In particular, the edges of the graph (which are in the worst case quadratic in the number of nodes) were represented using adjacency lists, matrices, etc.

From a theoretical perspective, the data structure used to represent the automata makes no difference whatsoever. From a practical perspective, however, this meant that model checkers could only handle automata with at most 10^3 to 10^6 reachable states [8]. Verification of most realistic systems was beyond the capability of such explicit-state engines. For example, a CPU with a single 32-bit register has more than $2^{32} \approx 4 \times 10^9$ possible states. Practical hardware verification via model checking had to wait for another breakthrough.

S. Chaki (✉)
Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA
e-mail: chaki@sei.cmu.edu

A. Gurfinkel
University of Waterloo, Waterloo, ON, Canada

E.M. Clarke et al. (eds.), *Handbook of Model Checking*,
DOI 10.1007/978-3-319-10575-8_8

The breakthrough appeared in the form of BDD-based symbolic model checking [8]. This paradigm uses a data structure called binary decision diagrams, BDDs, (see Chap. 7 and [1, 6]). BDDs are used to symbolically represent the transition relation of the automata or Kripke structures under analysis, and sets of states manipulated by the model-checking algorithm. Representing and manipulating transition relations and sets of states is sufficient for implementing model-checking algorithms for a wide range of temporal logics, for example, using the fixed-point construction described in Chap. 2.

Since its inception, BDD-based symbolic model checking has revolutionized formal verification and formal methods in profound ways. First, it has enabled practical verification of industrial systems—beginning with hardware [7] and extending to software [3]. Second, it has led to important developments for BDDs, such as new types of BDDs [22], variable-ordering heuristics [2, 24] and efficient implementations [31]. Finally, it has paved the way to other forms of symbolic model checking, especially those using efficient SAT solvers [4] and interpolants [21].

In this chapter, we present some of the key concepts and techniques in the area of BDD-based symbolic model checking. More specifically, we use reduced ordered BDDs (or ROBDDs). Unless otherwise mentioned, we use BDD to mean ROBDD. The goal of this chapter is not to be a comprehensive exposition of this rich and well-studied research area. Instead, we wish to present the basic ideas and algorithms to help someone unfamiliar with this topic get started, and to cite resources for the interested reader to find out more.

The rest of this chapter is organized as follows. Section 8.2 presents preliminary definitions borrowed from other chapters in the book. Section 8.3 presents basic concepts used in the rest of the chapter. Section 8.4 represents symbolic model checking of Kripke structures for CTL, fair CTL, and LTL. Section 8.5 presents symbolic model checking of reachability properties of push-down systems represented as Boolean programs. Section 8.6 concludes the chapter.

8.2 Preliminaries

Binary decision diagrams (BDDs) and their related concepts such as variable ordering and operations are presented in detail in Chap. 7. Therefore, we only give a brief overview of the BDD concepts and notation that are used in the rest of this chapter. Throughout, we assume that BDDs are ordered with respect to a fixed variable ordering and are reduced.

For set X, we write $\mathscr{P}(X)$ to mean the powerset of X. For a propositional formula f, let $Var(f)$ be the set of variables (a.k.a. atomic propositions) appearing in f. We assume that the reader is familiar with the basic concepts of temporal logic and its model-checking algorithms (see Chap. 2) and basic BDD operations (see Chap. 7). However, we use different notation and, for that reason, repeat some of the key definitions here. We refer the reader to earlier chapters for a more in-depth presentation of these topics.

Table 1 Correspondence between notations used in this chapter and Chap. 7

Name	Notation in this chapter	Notation in Chap. 7
Negation	$\neg\mathbf{f}$	NOT($\mathbf{f}$)
Conjunction	$\mathbf{f} \wedge \mathbf{g}$	AND($\mathbf{f}, \mathbf{g}$)
Disjunction	$\mathbf{f} \vee \mathbf{g}$	OR($\mathbf{f}, \mathbf{g}$)
Existential Quantification	$\exists x_i \,.\, \mathbf{f}$	EXISTS($\mathbf{f}, i$)
Variable Substitution	$Bdd(f[v_1, \ldots, v_n / v'_1, \ldots, v'_n])$	COMPOSE($\mathbf{g}_2, 1, v'_1$), where $\mathbf{g}_n = \text{COMPOSE}(\mathbf{f}, n, v'_n)$ $\mathbf{g}_{n-1} = \text{COMPOSE}(\mathbf{g}_n, n-1, v'_{n-1})$ $\ldots$ $\mathbf{g}_2 = \text{COMPOSE}(\mathbf{g}_3, 2, v'_2)$

Definition 1 (BDD) Reduced ordered BDDs are canonical representations of Boolean propositional formulas. The BDD for a formula f, denoted by $Bdd(f)$, is a directed acyclic graph (DAG). Given two BDDs $Bdd(f)$ and $Bdd(g)$, there are efficient algorithms to compute BDDs for the following operations:

- negation, $\neg Bdd(f)$, computes $Bdd(\neg f)$
- conjunction, $Bdd(f) \wedge Bdd(g)$, computes $Bdd(f \wedge g)$
- disjunction, $Bdd(f) \vee Bdd(g)$, computes $Bdd(f \vee g)$
- projection, $\exists v \,.\, Bdd(f)$, computes $Bdd(\exists v \,.\, f)$ where $v \in Var(f)$.
- renaming, computes $Bdd(f[v_1, \ldots, v_n / v'_1, \ldots, v'_n])$, where $f[v_1, \ldots, v_n / v'_1, \ldots, v'_n]$ is the formula obtained by simultaneously replacing each occurrence of v_i in f with v'_i for $1 \leq i \leq n$.

These operations are explained in detail in Chap. 7. Table 1 summarizes the correspondence between the notation used in this chapter and that of Chap. 7. For simplicity of presentation, we abuse notation by using propositional connectives, such as $\wedge$ and $\vee$, both as logical connectives and as the corresponding BDD operations. For example, $f \wedge g$ stands for a conjunction of propositional formulas f and g, while $Bdd(f) \wedge Bdd(g)$ stands for the BDD operation AND( ), which constructs the canonical BDD for $f \wedge g$ directly from BDDs for f and g. Furthermore, we sometimes write $\mathbf{f}$ for $Bdd(f)$, $\mathbf{0}$ for $Bdd(\text{FALSE})$, and $\mathbf{1}$ for $Bdd(\text{TRUE})$.

Definition 2 (Kripke structure) Let AP be a finite set of atomic propositions. A Kripke structure is a triple (S, R, L) where S is a finite set of states, $R \subseteq S \times S$ is a transition relation, and $L : S \mapsto \mathscr{P}(AP)$ labels each state with a set of atomic propositions.

Definition 3 (Model checking) Given a Kripke structure $M = (S, R, L)$, a designated initial state $s_0 \in S$, and a temporal logic formula φ, the model-checking problem is to decide whether M is a model of φ, i.e., whether $M, s_0 \models \varphi$.

8.3 Binary Decision Diagrams: The Basics

Any model-checking algorithm for deciding $M \models \varphi$ must somehow represent the Kripke structure M and manipulate sets of its states. In *explicit-state* model checking, M is represented explicitly as a graph (with the edges represented via matrices, adjacency lists, etc.) and sets of states are represented using standard data structures for sets of vertices (e.g., lists, balanced trees, hash tables, etc.). In contrast, in *symbolic* model checking, both the transition relation of M and its sets of states are modeled by Boolean functions and are represented symbolically by propositional formulas. Operations over sets of states are performed as symbolic manipulation of the corresponding formulas. While it is possible to manipulate the formulas directly, for example, using their abstract syntax trees (ASTs), efficient algorithms require efficient data structures to represent the formulas compactly and manipulate them efficiently. In this section, we show how BDDs are used for this purpose in symbolic finite-state model checking.

In the rest of this section, we fix M to be a Kripke structure $M = (S, R, L)$ over k atomic propositions $AP = \{p_1, \ldots, p_k\}$.

8.3.1 Representing Sets and Relations

In this section, we present key concepts needed to represent sets of states and transition relations symbolically. We begin with the notion of the *characteristic function* of a set, which connects sets with logical formulas. Next, we show how characteristic functions are used to represent sets of states and transition relations symbolically.

8.3.1.1 Characteristic Function

Let $Q \subseteq S$ be a set of states. The characteristic function of Q, denoted by $[\![Q]\!]$, is a mapping from S to $\{\text{TRUE}, \text{FALSE}\}$ defined as follows:

$$[\![Q]\!](s) = \text{TRUE} \iff s \in Q \tag{1}$$

That is, $[\![Q]\!](s)$ is true if and only if s is an element of Q. A set of states Q is represented by a BDD corresponding to the characteristic function $[\![Q]\!]$.

A *literal* of an atomic proposition p is either p itself or its negation $\neg p$. A *minterm* over AP is a conjunction in which every atomic proposition in AP appears either positively or negatively. Formally, a minterm is a formula $l_1 \wedge \cdots \wedge l_k$, where l_i is a literal of p_i for $1 \leq i \leq k$. Since AP has k elements, there are 2^k minterms, each corresponding to a distinct subset of AP. Minterms form a basis of propositional formulas over AP. Every propositional formula can be written uniquely as a disjunction of all minterms that imply it.

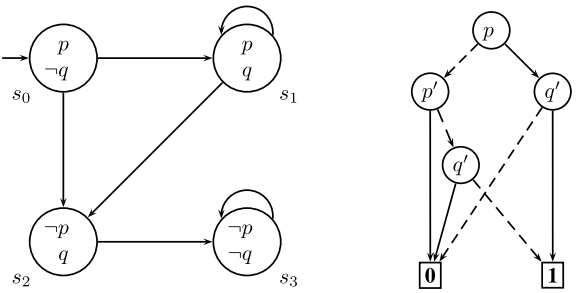


Fig. 1 A Kripke structure and a BDD for its transition relation (*dashed* and *solid lines* represent zero- and one-edges in the BDD, respectively)

8.3.1.2 Representing Sets

Without loss of generality, we assume that every state $s \in S$ is uniquely determined by the valuation of the atomic propositions in it. This implies that $|S| = 2^{|AP|}$. If this is not the case, then there are two possibilities. First, assume $|S| < 2^{|AP|}$; then we add additional states to S, but make sure they do not have any incoming or outgoing transitions. Second, assume $|S| > 2^{|AP|}$. Then, there are two distinct states, say $s, t \in S$, that agree on values of all atomic propositions. In this case, we add a new atomic proposition p to AP, and set p to true in s and to false in t.

Under the assumption above, a state $s \in S$ is uniquely determined by the set of literals true in s, where p is true in s if and only if $p \in L(s)$ and $\neg p$ is true in s if and only if $p \notin L(s)$. Formally, let χ be a mapping from S to minterms defined as follows:

$$\chi(s) = l_1 \wedge \cdots \wedge l_k \quad \text{where } l_i = \begin{cases} p_i & \text{if } p_i \in L(s) \\ \neg p_i & \text{otherwise} \end{cases} \tag{2}$$

Then, χ is a bijection.

For any $X \subseteq S$, the characteristic function $[\![X]\!]$ is represented by the propositional formula $f(X)$ defined as follows:

$$f(X) = \bigvee_{s \in X} \chi(s) \tag{3}$$

That is, $s \in X$ if and only if $\chi(s) \implies f(X)$. X is represented symbolically by a BDD for $f(X)$. We use $Bdd(X)$ (and $\mathbf{X}$) to refer to $Bdd(f(X))$ as long as the meaning is clear from the context.

Example 1 Consider the Kripke structure shown in Fig. 1. The corresponding function χ from states to minterms is

$$\chi(s_0) = p \wedge \neg q, \quad \chi(s_1) = p \wedge q, \quad \chi(s_2) = \neg p \wedge q, \quad \chi(s_3) = \neg p \wedge \neg q$$

Let $X = \emptyset$, $Y = \{s_0, s_1\}$, and $Z = \{s_1, s_3\}$. Then, the symbolic representation of the corresponding characteristic functions are:

$$f(X) = \text{FALSE}, \qquad f(Y) = (p \wedge \neg q) \vee (p \wedge q) = p,$$
$$f(Z) = (p \wedge q) \vee (\neg p \wedge \neg q)$$ □

8.3.1.3 Representing Relations

To represent the transition relation $R \subseteq S \times S$, we introduce a fresh set of atomic propositions $AP' = \{p'_1, \ldots, p'_k\}$. For a formula f over AP, we write $Prime(f)$ for the formula $f[p_1, \ldots, p_k / p'_1, \ldots, p'_k]$, and $UnPrime(f)$ for the formula $f[p'_1, \ldots, p'_k / p_1, \ldots, p_k]$. These operations are lifted to BDDs using a BDD renaming operation. We now extend χ to a bijection between $S \times S$ and the set of minterms over $AP \cup AP'$ as follows:

$$\chi(s,t) = \chi(s) \wedge Prime\big(\chi(t)\big) \tag{4}$$

For any $R \subseteq S \times S$, the characteristic function $[\![R]\!]$ is represented by a propositional formula $f(R)$ as follows:

$$f(R) = \bigvee_{(s,t) \in R} \chi(s,t) \tag{5}$$

That is, $(s,t) \in R$ if and only if $\chi(s,t) \implies f(R)$. R is represented symbolically using the BDD for $f(R)$. We use $Bdd(R)$ (and $\mathbf{R}$) to refer to $Bdd(f(R))$ as long as the meaning is clear from the context.

Example 2 Consider again the Kripke structure shown in Fig. 1. The symbolic representation of its transition relation is shown below:

$$\begin{aligned} f(R) &= \big(p \wedge \neg q \wedge p' \wedge q'\big) \vee \big(p \wedge \neg q \wedge \neg p' \wedge q'\big) \vee \big(p \wedge q \wedge p' \wedge q'\big) \\ &\quad \vee \big(p \wedge q \wedge \neg p' \wedge q'\big) \vee \big(\neg p \wedge q \wedge \neg p' \wedge \neg q'\big) \vee \big(\neg p \wedge \neg q \wedge \neg p' \wedge \neg q'\big) \\ &= \big(p \wedge q'\big) \vee \big(\neg p \wedge \neg p' \wedge \neg q'\big) \end{aligned}$$

The BDD $\mathbf{R}$ for the transition relation is shown in Fig. 1. □

8.3.2 Image Computation

Image and pre-image computations—that is, computing the set of successors or predecessors of a set of states, respectively—are the basic operations in any model-checking algorithm (see Chap. 2 for more details). In this section, we show how to implement this operation symbolically using the set and relation representations described above.

Given a set of states S, the *image* of S under the transition relation R is denoted by $Image(S, R)$ and is defined as follows:

$$Image(S, R) = \{t \mid \exists s \in S \,.\, (s, t) \in R\} \tag{6}$$

Intuitively, $Image(S, R)$ is the set of states reachable from S by one step of the transition relation R. Let $\mathbf{S}$ and $\mathbf{R}$ be the symbolic representations of S and R by BDDs over AP and $AP \cup AP'$, respectively. Then, the symbolic representation of $Image(S, R)$, denoted $\textsc{BddImage}(\mathbf{S}, \mathbf{R})$, is computed as follows:

$$\textsc{BddImage}(\mathbf{S}, \mathbf{R}) = UnPrime(\exists AP \,.\, \mathbf{S} \wedge \mathbf{R}) \tag{7}$$

The computation first constructs the BDD for the conjunction of S and R, then projects away using existential quantification all of the pre-state variables AP, and finally renames the result from AP' to AP variables.

Example 3 As an example, consider the computation of $\textsc{BddImage}(\mathbf{S}, \mathbf{R})$, where R is the transition relation of the Kripke structure in Fig. 1, and $S = \{s_0\}$:

$$\begin{aligned}
&\textsc{BddImage}(\mathbf{S}, \mathbf{R}) \\
&\quad = UnPrime\big(\exists p, q \,.\, (p \wedge \neg q) \wedge \big((p \wedge q') \vee (\neg p \wedge \neg p' \wedge \neg q')\big)\big) \\
&\quad = UnPrime\big(\exists p, q \,.\, (p \wedge \neg q) \wedge (p \wedge q')\big) \\
&\quad = UnPrime\big(q'\big) = q = Bdd\big(f\big(\{s_1, s_2\}\big)\big)
\end{aligned}$$

Similarly, the pre-image of S under R is denoted by $PreImage(S, R)$ and defined as follows:

$$PreImage(T, R) = \{s \mid \exists t \in T \,.\, (s, t) \in R\} \tag{8}$$

Intuitively, $PreImage(S, R)$ is the set of all states that can reach a state in S by one step of the transition relation R. Given symbolic representations $\mathbf{T}$ and $\mathbf{R}$, of T and R respectively, the symbolic representation of $PreImage(T, R)$, denoted $\textsc{BddPreImage}(\mathbf{T}, \mathbf{R})$, is computed as follows:

$$\textsc{BddPreImage}(\mathbf{T}, \mathbf{R}) = \exists AP' \,.\, \mathbf{R} \wedge Prime(\mathbf{T}) \tag{9}$$

That is, the BDD for T is first renamed to be over post-state variables AP', then it is conjoined with the BDD R for the transition relation, and finally all post-state variables are quantified out existentially.

Example 4 As an example, consider the computation of $PreImage(T, R)$, where R is the transition relation of the Kripke structure in Fig. 1, and $T = \{s_1, s_2\}$:

$$\begin{aligned}
&\textsc{BddPreImage}(\mathbf{T}, \mathbf{R}) \\
&= \exists p', q' \,.\, \big((p \wedge q') \vee (\neg p \wedge \neg p' \wedge \neg q')\big) \wedge Prime(q) \\
&= \exists p', q' \,.\, (p \wedge q') \wedge q' \\
&= p = Bdd\big(f\big(\{s_0, s_1\}\big)\big)
\end{aligned}$$

Efficient implementations of BddImage and BddPreImage combine the conjunction and existential quantification operations together in a single operation called a *relational product*. In practice, the relational product is the bottleneck for BDD-based model-checking algorithms. Even when the input and output of the image computation are manageable, the intermediate results computed during the relational product often explode. Further details on relational product and its implementation are presented in Chap. 7.

8.3.3 Partitioned Transition Relation

In the previous section, we have shown how to compute the image and pre-image of a transition relation represented by a single BDD. This is called *monolithic* image computation. In practice, often even when R, S, and $Image(S, R)$ have efficient BDD representations, the intermediate result is very large. This is often referred to as the "hump" of the image computation.

In this case, it is desirable to partition the transition relation R into a set of BDDs and operate directly on such a partitioned relation. In this section, we show two main techniques called *disjunctive* and *conjunctive* decomposition, respectively.

8.3.3.1 Disjunctive Decomposition

Disjunctive decomposition is the simpler of the two. It is based on the fact that existential quantification distributes over disjunction, i.e.,

$$\exists X \,.\, A \vee B \iff (\exists X \,.\, A) \vee (\exists X \,.\, B) \tag{10}$$

Specifically, assume that R is represented by a set of BDDs $\mathbf{R}_1, \ldots, \mathbf{R}_n$ such that

$$\mathbf{R} = \bigvee_{1 \le i \le n} \mathbf{R}_i \tag{11}$$

This is often the case when R is the result of asynchronous composition of several transition relations. Then, since both existential quantification and variable renaming distribute over disjunction, we have

$$\begin{aligned}\textsc{BddImage}(\mathbf{S},\mathbf{R}) &= \mathit{UnPrime}(\exists A P \,.\, \mathbf{S} \wedge \mathbf{R}) \\ &= \mathit{UnPrime}\left(\exists A P \,.\, \mathbf{S} \wedge \left(\bigvee_{1\le i\le n} \mathbf{R}_i\right)\right) \\ &= \bigvee_{1\le i\le n} \underbrace{\mathit{UnPrime}(\exists A P \,.\, \mathbf{S} \wedge \mathbf{R}_i)}_{\text{done one } \mathbf{R}_i \text{ at a time}}\end{aligned}$$

The computation is done similarly for the BddPreImage:

$$\begin{aligned}\textsc{BddPreImage}(\mathbf{S},\mathbf{R}) &= \exists A P' \,.\, \mathit{Prime}(\mathbf{S}) \wedge \mathbf{R} \\ &= \exists A P' \,.\, \mathit{Prime}(\mathbf{S}) \wedge \left(\bigvee_{1\le i\le n} \mathbf{R}_i\right) \\ &= \bigvee_{1\le i\le n} \underbrace{\exists A P' \,.\, \mathit{Prime}(\mathbf{S}) \wedge \mathbf{R}_i}_{\text{done one } \mathbf{R}_i \text{ at a time}}\end{aligned}$$

The advantage of such modular image and pre-image computation is that, in practice, it often leads to intermediate BDDs of smaller size.

8.3.3.2 Conjunctive Decomposition

Conjunctive decomposition is based on the principle of *early quantification* [18]. Let X be a set of variables, and A and B be two propositional formulas such that $X \cap \mathit{var}(A) = \emptyset$. Then,

$$\exists X \,.\, A \wedge B \iff A \wedge \exists X \,.\, B \tag{12}$$

That is, since A has no variables in X, the existential quantification can be "pushed in".

Specifically, assume that R is represented by a set of BDDs $\mathbf{R}_1, \ldots, \mathbf{R}_n$ such that

$$\mathbf{R} = \bigwedge_{1\le i\le n} \mathbf{R}_i \tag{13}$$

This is often the case when R is the result of synchronous composition of several transition relations. Then,

$$\begin{aligned}\textsc{BddImage}(\mathbf{S},\mathbf{R}) &= \mathit{UnPrime}(\exists AP \,.\, \mathbf{S} \wedge \mathbf{R}) \\ &= \mathit{UnPrime}\left(\exists AP \,.\, \mathbf{S} \wedge \left(\bigwedge_{1\le i\le n} \mathbf{R}_i\right)\right) \\ &= \mathit{UnPrime}\left(\exists V_1 \,.\, \left(\exists V_2 \,.\, \ldots \exists V_n \,.\, (\mathbf{S} \wedge \mathbf{R}_n) \cdots \wedge \mathbf{R}_2\right) \wedge \mathbf{R}_1\right)\end{aligned}$$

where, for $1 \le i \le n$,

$$V_i = AP \cap \left(\mathit{Var}(\mathbf{R}_i) \setminus \bigcup_{1\le j<i} \mathit{Var}(\mathbf{R}_j)\right)$$

That is, V_i is the set of atomic propositions p such that i is the smallest value for which p appears in $\mathbf{R}_i$.

Similarly, for the BddPreImage

$$\begin{aligned}\textsc{BddPreImage}(\mathbf{S},\mathbf{R}) &= \exists AP' \,.\, \mathit{Prime}(\mathbf{S}) \wedge \mathbf{R} \\ &= \exists AP' \,.\, \mathit{Prime}(\mathbf{S}) \wedge \left(\bigwedge_{1\le i\le n} \mathbf{R}_i\right) \\ &= \exists V_1' \,.\, \left(\exists V_2' \,.\, \cdots \exists V_n' \,.\, \left(\mathit{Prime}(\mathbf{S}) \wedge \mathbf{R}_n\right) \cdots \wedge \mathbf{R}_2\right) \wedge \mathbf{R}_1\end{aligned}$$

where, for $1 \le i \le n$,

$$V_i' = AP' \cap \left(\mathit{Var}(\mathbf{R}_i) \setminus \bigcup_{1\le j<i} \mathit{Var}(\mathbf{R}_j)\right)$$

That is, V_i is the set of primed atomic propositions p such that i is the smallest value for which p appears in $\mathbf{R}_i$. Once again, the advantage of early quantification is that, in practice, it often leads to intermediate BDDs of smaller size.

8.3.4 Historical Perspective

Binary decision diagrams (more specifically, binary decision programs) were introduced by Lee [19]. The key idea here was to build on the encoding proposed by Shannon [30] to give an alternative (and superior) representation of switching circuits. The representation was a program consisting of a sequence of "two-address conditional transfer instructions". Such a program is equivalent to a BDD, where each instruction corresponds to a node. BDDs were subsequently popularized by Akers [1] and Boute [5]. Bryant [6] facilitated the use of BDDs in efficient automated verification by proposing and developing two ideas—fixed variable ordering and shared sub-graphs—that lead to a compressed canonical form.

The use of symbolic representations for verification has a long history. For example, Coudert, Berthet, and Madre [17] present a symbolic algorithm to check equivalence between two deterministic Moore machines. They perform a breadth-first traversal of the product state machine. States and transitions are represented symbolically using BDDs, which the authors refer to as "typed decision graphs".

Touati, Savoj, Lin, Brayton, and Sangiovanni-Vincentelli [32] build on the work by Coudert et al. [17] to develop more efficient algorithms for image computation using BDDs. In particular, they propose a new variable-ordering heuristic to control BDD sizes, and a new image computation algorithm that leverages the conjunctive structure of the transition relation.

Burch, Clarke, McMillan, Dill, and Hwang [8] present symbolic model-checking algorithms for the μ-calculus [25] using BDDs, their implementation, and experimental results demonstrating successful verification of systems with 10^{20} states. This contrasts with model checkers that use explicit-state enumeration techniques, and are reported to scale to systems with 10^6 reachable states only.

The problem of computing a good BDD variable ordering has received wide attention since it is crucial to keeping the BDD sizes manageable, and therefore to the success of BDD-based model checking. State-of-the-art BDD packages [31] use dynamic variable ordering, where the variables are reordered on the fly using heuristics designed to minimize BDD sizes. A popular minimization heuristic is "sifting", which was introduced by Rudell [29].

A related, and also widely studied, problem is "conjunction scheduling" or "clustering", i.e., finding a good conjunctive partitioning of the transition relation and then ordering the partitions during image computation so that BDD sizes remain small. Moon, Hachtel, and Somenzi [23] present an algorithm to solve this problem that uses information about the dependence matrix of the transition relation and its permutation.

Ranjan, Aziz, Brayton, Plessier, and Pixley [26] address both the variable-ordering and clustering problems. First, they present a dynamic variable-ordering technique that is parameterized by the total number of BDD nodes at which reordering starts, and the minimum increase in BDD size between two successive reordering invocations by the BDD manager. These parameters are tuned at runtime, leading to a highly adaptive reordering scheme. For clustering, the latch transition relations are first ordered heuristically. Second, they are ordered such that the product of the transition relation BDD sizes in each cluster is below a user-specified threshold.

Another problem studied in BDD-based verification is approximate finite state machine (FSM) traversal, whereby the transition relation of the FSM is first decomposed into several sub-relations, then each sub-relation is traversed independently, and finally the results are combined together. In general, this results in an over-approximation of the actual result, since the overall process can be viewed as a type of *Cartesian abstraction*. Cho, Hachtel, Macii, Plessier, and Somenzi [12] present and evaluate several heuristics for finding good decompositions and effective traversal strategies.

Ravi and Somenzi [27] have explored BDD-based symbolic state space exploration algorithms that aim to compute a subset of the actual set of reachable states.

Combined with the approximate FSM traversal algorithms described above, this enables both an upper and a lower bound on the reachable states to be obtained. The key idea used in this paper is that of "BDD density" which is defined to be the ratio of the number of minterms of the BDD and the number of its nodes. The algorithm attempts to always maintain a BDD with high density. If the BDD size becomes too large during state space exploration, it is under-approximated to obtain a high-density BDD, and the exploration continues with the resulting BDD. The authors also present a state space traversal strategy that combines breadth-first and depth-first strategies in order to maximize BDD density.

Cabodi, Camurati, and Quer [9] explore the computation and use of "activity profiles" of BDDs. Informally, this measures, for each BDD node, its level of activity in terms of time and memory usage. The profile is computed by using an inexpensive reachability analysis as a learning phase. It is subsequently used to improve partial state space traversals that use transition relation subsetting and a combination of breadth-first and depth-first exploration [27].

Cabodi, Camurati and Quer [10] also explore dynamic BDD-partitioning strategies to optimize complex operations used in state space exploration. The key idea behind this partitioning is to recursively find splitting variables that lead to disjoint subsets, well-balanced partitions, and overall minimized BDD size. In addition, the authors also experimentally characterize the cost of complex BDD operations.

More recently, Xu, Williams, Mony, and Baumgartner [33] explore the use of automated netlist-based hint generation to improve scalability of BDD-based reachability analysis. Hints are used during reachability analysis to constrain the BDDs and guide the state space exploration. They often lead to smaller peak BDD sizes. Completeness is ensured by restoring the original transition relation once all hints have been used.

Despite their long history, BDD-based symbolic verification techniques continue to be an active area of applied research. A recent report on the Hardware Model Checking Competition (HWMCC'14), in which model-checking tools compete on verification problems from the hardware domain, shows that all of the top portfolio-based tools employ at least one BDD-based model-checking algorithm [11].

8.4 Model Checking Kripke Structures

In this section, we present BDD-based symbolic model-checking algorithms for CTL and LTL temporal logics. Throughout the section, we assume that M is a Kripke structure (S, R, L) over a set of atomic propositions AP; R is represented symbolically using a BDD, and there is a designated initial state $s_0 \in S$ of M represented by the BDD $\mathbf{s_0}$.

```
1: function CHECKSAFETY(M, s_0, AG p)                ▷ M = (S, R, L), s_0 ∈ S
2:     S := 0                                        ▷ set of reachable states is initially empty
3:     F := s_0                                      ▷ frontier set of states to be explored next
4:     do
5:         S := S ∨ F                                ▷ update reachable states
6:         F := BDDIMAGE(F, R) ∧ ¬S                  ▷ update frontier
7:     while F ≠ 0
8:     return (S ∧ ¬p) = 0                           ▷ check that all reachable states are labeled with p
```

Fig. 2 An algorithm to decide whether $M \models \mathbf{AG}\, p$

8.4.1 Reachability/Invariant/AG

We begin with the simple case of model checking a *safety* property φ. Model checking of any safety property can be reduced to model checking a CTL formula of the form $\mathbf{AG}\, p$, where p is a single atomic proposition. This follows from the fact that violation of any safety property can be reduced to reachability of some well-defined bad state (see Chap. 2). If $M \models \varphi$, then φ is called an *invariant* of M.

The algorithm for deciding whether $M \models \varphi$ is shown in Fig. 2. It works by iteratively computing the set of states reachable from the initial state s_0 (lines 2–7), and then checking whether there exists a reachable state that does not satisfy p (line 8). Throughout the i-th iteration of the main loop, it maintains two variables, **S**—a BDD representing the set of states reached in fewer than i steps, and **F**—the frontier—a BDD representing states reachable in exactly i steps. Note that the forward image is only applied to the frontier **F**, and not to all the states discovered so far. This helps to alleviate the bottleneck of the relational product computation.

The termination of the algorithm follows from the fact that the state space of M is finite, and the set **S** is increased in each iteration of the loop. The number of iterations is bounded by the number of (reachable) states in M.

8.4.2 CTL Model Checking

We now present the algorithm for model checking arbitrary Computation Tree Logic (CTL) formulas. We refer the reader to Chap. 2 for a thorough presentation of CTL and other temporal logics. In this chapter we use only the minimal amount of material about temporal logics necessary for completeness. Without loss of generality, we restrict the syntax of CTL as follows, where φ, φ_1, and φ_2 are CTL formulas and $p \in AP$:

$$\varphi = p \mid \neg\varphi_1 \mid \varphi_1 \wedge \varphi_2 \mid \mathbf{EX}\varphi_1 \mid \mathbf{EG}\varphi_1 \mid \mathbf{E}(\varphi_1\, \mathbf{U}\, \varphi_2) \tag{14}$$

We use $\bot$ and $\top$ as shorthands for $(p_0 \wedge \neg p_0)$ and $\neg\bot$, respectively, where $p_0 \in AP$ is a distinguished atomic proposition. Other operators are replaced using standard equivalences (see [16] and Chap. 2). For example,

$$\mathbf{EF}\varphi = \mathbf{E}(\top\, \mathbf{U}\, \varphi), \qquad \mathbf{AG}\varphi = \neg\mathbf{EF}\neg\varphi$$

```
1: function CHECKCTL(M, s_0, φ)                          ▷ M = (S, R, L), s_0 ∈ krst
2:     S = CTLSTATES(M, φ)                               ▷ compute set of states satisfying φ
3:     return (s_0 ∧ ¬S) = 0                             ▷ check that the initial state satisfies φ

4: function CTLSTATES(M, φ)
5:     if φ = p then return p
6:     else if φ = ¬φ_1 then return ¬CTLSTATES(M, φ_1)
7:     else if φ = φ_1 ∧ φ_2 then return CTLSTATES(M, φ_1) ∧ CTLSTATES(M, φ_2)
8:     else if φ = EXφ_1 then
9:         return EXSTATES(M, CTLSTATES(M, φ_1))
10:    else if φ = EGφ_1 then
11:        return EGSTATES(M, CTLSTATES(M, φ_1))
12:    else if φ = E(φ_1 U φ_2) then
13:        return EUSTATES(M, CTLSTATES(M, φ_1), CTLSTATES(M, φ_2))

14: function EXSTATES(M, S_φ)                            ▷ M = (S, R, L)
15:    return BDDPREIMAGE(S_φ, R)

16: function EGSTATES(M, S_φ)                            ▷ M = (S, R, L)
17:    S' := S_φ                                         ▷ initialize
18:    do
19:        S := S'                                       ▷ store previous result
20:        S' := S_φ ∧ BDDPREIMAGE(S, R)                 ▷ compute new result
21:    while S' ≠ S                                      ▷ stop when fixed point is reached
22:    return S

23: function EUSTATES(M, S_1, S_2)                       ▷ M = (S, R, L)
24:    S' := 0
25:    do
26:        S := S'                                       ▷ store previous result
27:        S' := S_2 ∨ (S_1 ∧ BDDPREIMAGE(S, R))         ▷ update new result
28:    while S' ≠ S                                      ▷ stop when fixed point is reached
29:    return S
```

Fig. 3 CTL model-checking algorithm

Recall from Chap. 2 that every CTL formula φ is a *state formula*, and that $M \models \varphi$ if and only if the initial state s_0 of M satisfies φ. An algorithm CHECKCTL to decide whether $M \models \varphi$ is given in Fig. 3. It works in two steps:

- use an auxiliary function CTLSTATES to compute the set of states S of M that satisfy φ;
- return true if and only if $s_0 \in S$.

Note that throughout the algorithm sets of states and the transition relation are represented by BDDs and are manipulated by BDD operations. Recall that we write $\mathbf{S}$ for a BDD representing propositional formula S and that we do not distinguish between sets of states of a Kripke structure and their characteristic propositional formulas. Function CTLSTATES works differently based on the syntax of its argument CTL formula φ. In particular, it uses functions EXSTATES, EGSTATES, and EUSTATES to handle the cases where φ is of the form $\mathbf{EX}\varphi_1$, $\mathbf{EG}\varphi_1$ and $\mathbf{E}(\varphi_1\ \mathbf{U}\ \varphi_2)$,

respectively. Function EXSTATES(M, S) computes the set of predecessors of S using *PreImage* computation described in Sect. 8.3.2. Function EGSTATES(M, S) iteratively computes the set of states from which there exists a path consisting only of states from S. This is a greatest fixed point (cf. Chap. 2) computation. Let S_i be the value of S in the i-th iteration of the algorithm. Initially, S_0 contains all states. Intuitively, the algorithm assumes that all states in S have a successor in S. In the first iteration, S_1 is set to the set of all states in S_0 that have at least one immediate successor in S_0 (and states that have no successors in S_0 are removed). Following this reasoning, in iteration i, S_i contains all states that have a path of length $i-1$ in S_{i-1}. When the algorithm terminates, S contains only states that have an infinite path contained in S. The set S is reduced in each iteration, and since it is finite initially, the algorithm eventually terminates. Function EUSTATES(M, S_1, S_2) computes the set of states from which there exists a path on which states from S_1 appear initially until a state from S_2 appears. This is a least fixed point (cf. Chap. 2) computation similar to the reachability computation described in Sect. 8.4.1. Note that we use a *strong* version (cf. Chap. 2) of the until-operator $\mathbf{E}(\varphi_1 \mathbf{U} \varphi_2)$ that requires φ_2 to hold in the last state of satisfiable computation. An alternative definition of *weak* until (cf. Chap. 2), also allows for computations on which φ_1 holds forever without ever reaching a state satisfying φ_2. Note that the efficiency of the termination checks (line 21 of EGSTATES and line 28 of EUSTATES) is due to the canonicity property of BDDs.

The function CTLSTATES is recursive. Termination is guaranteed by the fact that recursive calls always work on syntactically smaller formulas. Furthermore, since a formula may contain the same sub-formula multiple times, results can be cached to improve efficiency.

8.4.3 Fair CTL Model Checking

In this section, we extend the symbolic CTL model-checking algorithm to handle CTL with additional fairness constraints. A more detailed presentation of fairness is available in Chap. 2. We assume that the fairness constraints are given by a set of CTL formulas $\mathscr{F} = \{\phi_1, \ldots, \phi_n\}$. The pseudo-code for the model-checking algorithm CHECKFAIRCTL for $M \models \varphi$ under $\mathscr{F}$ is given in Fig. 4. It works in several steps:

1. function FAIRSTATES computes the set S_F of all fair states—states from which there is at least one path that satisfies all the fairness conditions infinitely often;
2. function CTLFAIRSTATES computes the set of states S_φ of M that satisfy φ under $\mathscr{F}$;
3. finally, the algorithm returns true if and only if the initial state s_0 is in $S_F \cap S_\varphi$.

```
1: var S_F                                              ▷ a BDD representing the set of all fair states
2: function CHECKFAIRCTL(M, s_0, φ, ℱ)                  ▷ M = (S, R, L), s_0 ∈ S, ℱ = {φ_1, ..., φ_n}
3:     S_F := FAIRSTATES(M, ℱ)                          ▷ the set of all fair states
4:     S_φ := CTLFAIRSTATES(M, φ, ℱ)                    ▷ compute set of states satisfying φ under ℱ
5:     S_φ := S_φ ∧ S_F                                 ▷ restrict to fair states
6:     return (s_0 ∧ ¬S_φ) = 0                          ▷ check that all initial states satisfy φ under ℱ

7: function FAIRSTATES(M, ℱ)
8:     return EGFAIRSTATES(M, TRUE, ℱ)

9: function CTLFAIRSTATES(M, φ, ℱ)                      ▷ M = (S, R, L), ℱ = {φ_1, ..., φ_n}
10:    if φ = p then
11:        return p
12:    else if φ = ¬φ_1 then
13:        return ¬CTLFAIRSTATES(M, φ_1, ℱ)
14:    else if φ = φ_1 ∧ φ_2 then
15:        return CTLFAIRSTATES(M, φ_1, ℱ) ∧ CTLFAIRSTATES(M, φ_2, ℱ)
16:    else if φ = EXφ_1 then
17:        S_1 := CTLFAIRSTATES(M, φ_1)
18:        return EXSTATES(M, S_1 ∧ S_F)
19:    else if φ = EGφ_1 then
20:        return EGFAIRSTATES(M, φ_1, ℱ)
21:    else if φ = E(φ_1 U φ_2) then
22:        S_1 := CTLFAIRSTATES(M, φ_1)
23:        S_2 := CTLFAIRSTATES(M, φ_2)
24:        return EUSTATES(M, S_1, S_2 ∧ S_F)

25: function EGFAIRSTATES(M, φ, ℱ)                      ▷ M = (S, R, L)
26:    S_1 := CTLSTATES(M, φ)                           ▷ states satisfying φ
27:    Q' := S_1                                        ▷ initially, assume all states satisfy EGφ
28:    do                                               ▷ repeat until fixed point
29:        Q := Q'                                      ▷ store previous result
30:        Q' := 1
31:        for i = 1 → n do
32:            F_i := CTLSTATES(M, φ_i)
33:            EU_i := EUSTATES(M, S_1, Q ∧ F_i)
34:            Q' := Q' ∧ EXSTATES(M, EU_i)
35:    while Q ≠ Q'
36:    return Q
```

Fig. 4 An algorithm for CTL model checking under fairness constraints. Functions CTLSTATES, EXSTATES, and EUSTATES are from Fig. 3

8.4.3.1 Function EGFAIRSTATES

This function is the key part of the algorithm. Recall that $\mathscr{F} = \{\phi_1, \ldots, \phi_n\}$. CHECKFAIREG$(M, \varphi, \mathscr{F})$ computes the set of all states that satisfy $EG\varphi$ under the fairness condition $\mathscr{F}$. That is, it returns the set of all states that start an infinite path π such that on π every fairness condition in $\mathscr{F}$ is satisfied infinitely often, and every state on π satisfies φ. Note that in the case $\varphi =$ TRUE, the function returns the set of all fair states. This observation is used to implement FAIRSTATES.

EGFAIRSTATES is implemented as an iterative computation of the greatest fixed point of the function $\lambda : \mathscr{P}(S) \mapsto \mathscr{P}(S)$ defined as follows:

$$\lambda(Z) = \varphi \wedge \bigwedge_{i=1}^{n} \mathbf{EXE}\big(\varphi \,\mathbf{U}\, (Z \wedge \phi_i)\big) \tag{15}$$

In the above, we use CTL formulas and sets of states that satisfy them (without fairness) interchangeably. Intuitively, a state s is in the greatest fixed point of $\lambda(Z)$ if and only if for each fairness condition ϕ_i it can reach a state s_i satisfying ϕ_i, and each s_i, in turn, can do the same. We refer the reader to Chap. 26 for further details about the μ-calculus, which is necessary to understand the correctness of (15) and CHECKFAIREG, but is beyond the scope of this chapter. The formal correctness of CHECKFAIREG is also presented in [16].

8.4.3.2 Function CTLFAIRSTATES

CTLFAIRSTATES works based on the structure of the input CTL formula φ. For atomic formulas and propositional connectives, it works exactly as CTLSTATES. For φ of the form $\mathbf{EX}\varphi_1$ it computes the set of predecessors of the fair subset of the states that satisfy φ_1. This follows from the fact [16] that the set of states satisfying $\mathbf{EX}\varphi_1$ under $\mathscr{F}$ is equivalent to the set of states that have a successor s which (i) satisfies $\mathbf{EX}\varphi_1$ without fairness, and (ii) belong to S_F.

The case where φ is of the form $\mathbf{E}(\varphi_1 \,\mathbf{U}\, \varphi_2)$ follows from the fact that the set of states satisfying φ is also the set of states from which there is a path on which states satisfying φ_1 (without fairness) appear initially until a state appears that both (i) satisfies φ_2 (without fairness), and (ii) belongs to S_F.

8.4.4 LTL Model Checking

In this section, we present a BDD-based symbolic model-checking algorithm for LTL formulas. The explicit-state LTL model-checking algorithm [20] for deciding $M, s_0 \models \varphi$ works by (i) constructing a Büchi automaton B_φ that represents $\neg\varphi$, and (ii) checking for an accepting run in the Büchi automaton $B_{M,\varphi}$ obtained by composing M and B_φ. All of the automata—M, B_φ, and $B_{M,\varphi}$—are represented explicitly as graphs. Further details about this algorithm, and about automata-theoretic model checking in general, are presented in Chap. 4.

The symbolic version of the LTL model-checking algorithm [15] is similar, but with the following important exceptions. First, instead of an explicit Büchi automaton for $\neg\varphi$, it constructs a symbolic tableau, i.e., a Kripke structure M_φ with a set of fairness conditions $\mathscr{F}_\varphi$ such that the set of all fair executions of M_φ is exactly the set of all computations that satisfy $\neg\varphi$. Second, it composes the tableau M_φ with the input Kripke structure M symbolically by conjoining their transition relations.

1: **function** CHECKLTL(M, s_0, φ) ▷ $M = (S, R, L), s_0 \in S, \varphi$ is an LTL formula
2: $M_\varphi, \mathscr{F}_\varphi := \text{LTLTABLEAU}(\neg\varphi)$
3: $M' := M \wedge M_\varphi$
4: $\mathbf{S} := \text{FAIRSTATES}(M', \mathscr{F}_\varphi)$
5: **return** $(\mathbf{s_0} \wedge \neg\mathbf{S}) = \mathbf{0}$

Fig. 5 Symbolic algorithm for LTL model checking

Third, it uses the FAIRSTATES algorithm from Sect. 8.4.3.1 (shown in Fig. 4) as a sub-routine to check whether the initial state of M is a fair state of the composed structure M'. If so (i.e., if the initial state of M is a fair state of M') then the overall algorithm terminates by declaring $M, s_0 \not\models \varphi$. This is correct because, in this case, there is an execution of M starting with s_0 that satisfies $\neg\varphi$. Otherwise, it terminates by declaring $M, s_0 \models \varphi$. This is correct because, in this case, all executions of M starting with s_0 satisfy φ. Note that the tableau can be viewed as a symbolic representation of a Büchi automaton. Further details on this topic can be found in Chap. 4 and in [28].

The algorithm for symbolic LTL model checking is shown in Fig. 5. The main new procedure is LTLTABLEAU(M, φ), which constructs the Kripke structure M_φ and the fairness constraint $\mathscr{F}_\varphi$ symbolically using BDDs. Note that the transition relation of the product Kripke structure M' constructed at line 3 is represented symbolically by the BDD $\mathbf{R} \wedge \mathbf{R_T}$ where R is the transition relation of M and R_T is the transition relation of the tableau M_φ. We now discuss a way of implementing LTLTABLEAU inside a model checker.

8.4.4.1 Restricted Path Formula

We use the convention that every LTL formula φ is syntactically of the form $\mathbf{A}\Psi$ where Ψ is a *restricted path formula*, i.e., one in which the only state sub-formulas are Boolean combinations of atomic propositions. Formally, the syntax of Ψ is defined inductively as follows, where $p \in AP$ and Ψ_1 and Ψ_2 are also restricted path formulas:

$$\Psi = p \mid \neg\Psi_1 \mid \Psi_1 \wedge \Psi_2 \mid \mathbf{X}\Psi_1 \mid \Psi_1 \,\mathbf{U}\, \Psi_2 \tag{16}$$

Definition 4 (Elementary formula) Given a restricted path formula Ψ, the set of elementary formulas of Ψ is denoted by $el(\Psi)$ and defined inductively over the structure of Ψ as follows:

$$\begin{aligned} el(p) &= p \\ el(\neg\Psi_1) &= el(\Psi_1) \\ el(\Psi_1 \wedge \Psi_2) &= el(\Psi_1) \cup el(\Psi_2) \\ el(\mathbf{X}\Psi_1) &= \{\mathbf{X}\Psi_1\} \cup el(\Psi_1) \\ el(\Psi_1 \,\mathbf{U}\, \Psi_2) &= \left\{\mathbf{X}(\Psi_1 \,\mathbf{U}\, \Psi_2)\right\} \cup el(\Psi_1) \cup el(\Psi_2) \end{aligned}$$

Note that $el(\Psi)$ contains all atomic propositions appearing in Ψ.

8.4.4.2 Algorithm LTLTABLEAU

Let $M = (S, R, L)$ be a Kripke structure over the set of atomic propositions AP. We denote by $\mathbf{p}$ the set of atomic propositions appearing in Ψ. Clearly, $\mathbf{p} \subseteq AP$. Let $\mathbf{r} = AP \setminus \mathbf{p}$ be the set of atomic propositions appearing in M but not in Ψ. Let $\mathbf{p}'$ and $\mathbf{r}'$ be the primed versions of $\mathbf{p}$ and $\mathbf{r}$, respectively. Thus, the transition relation R of M is represented by a BDD over $\mathbf{p} \cup \mathbf{r} \cup \mathbf{p}' \cup \mathbf{r}'$.

Let the set of atomic propositions AP_T be the set of all elementary sub-formulas $el(\neg\varphi)$. Therefore, from Definition 4, we know that $AP_T = el(\varphi) \supseteq \mathbf{p}$. The function LTLTABLEAU(φ) constructs a *tableau* T from the formula φ as a Kripke structure (S_T, R_T, L_T) over the set of atomic propositions AP_T. Formally, the components of T are defined as follows:

- *States and Labeling Function:* Each state of T is a subset of AP_T. Thus, $S_T = \mathscr{P}(AP_T)$. The labeling function L_T is the identity mapping, i.e., each state is labeled by the set of propositions it corresponds to.
- *Transition Relation:* Let *sat* be a function from restricted path formulas to $\mathscr{P}(S_T)$ such that $sat(\varphi)$ is the set of all states of T that satisfy φ. Formally, for $\Psi \in el(\varphi)$, $sat(\Psi)$ is computed symbolically using BDDs by structural induction over Ψ as follows:

$$
\begin{aligned}
sat(g) &= Bdd(g) \text{ if } g \in el(\Psi) \\
sat(\neg\Psi_1) &= \neg sat(\Psi_1) \\
sat(\Psi_1 \wedge \Psi_2) &= sat(\Psi_1) \wedge sat(\Psi_2) \\
sat(\Psi_1 \,\mathbf{U}\, \Psi_2) &= sat(\Psi_2) \vee \big(sat(\Psi_1) \wedge Bdd\big(\mathbf{X}(\Psi_1 \,\mathbf{U}\, \Psi_2)\big)\big)
\end{aligned}
$$

  Note that by Definition 4, $\mathbf{X}(\Psi_1 \,\mathbf{U}\, \Psi_2)$ is an elementary formula. Let $\mathbf{q} = AP_T \setminus \mathbf{p}$ be the set of atomic propositions appearing in T but not in $\mathbf{p}$, and let $\mathbf{q}'$ be the set of primed versions of $\mathbf{q}$. The transition relation R_T of the tableau T is represented by the BDD over $\mathbf{p} \cup \mathbf{q} \cup \mathbf{p}' \cup \mathbf{q}'$ corresponding to the following propositional formula:

$$
R_T = \bigwedge_{\mathbf{X}\Psi' \in el(\Psi)} sat\big(\mathbf{X}\Psi'\big) \iff Prime\big(sat\big(\Psi'\big)\big) \tag{17}
$$

Fairness Constraint. Finally, the fairness constraint $\mathscr{F}_\varphi$ is given by the following set:

$$
\mathscr{F}_\varphi = \big\{ sat\big(\neg(\Psi_1 \,\mathbf{U}\, \Psi_2) \vee \Psi_2\big) \;\big|\; \Psi_1 \,\mathbf{U}\, \Psi_2 \text{ occurs in } \varphi \big\} \tag{18}
$$

Each element of $\mathscr{F}_\varphi$ is represented symbolically by a BDD.

We have thus shown how the tableau M_φ and the fairness constraints $\mathscr{F}_\varphi$ are constructed symbolically using BDDs. The correctness of the construction of M_φ, and the correctness of the overall symbolic LTL model-checking algorithm (shown in Fig. 5) by composing M_φ with M and checking for fair states (in accordance with $\mathscr{F}_\varphi$) in the resulting automata are presented in [16] and we will not describe them further here.

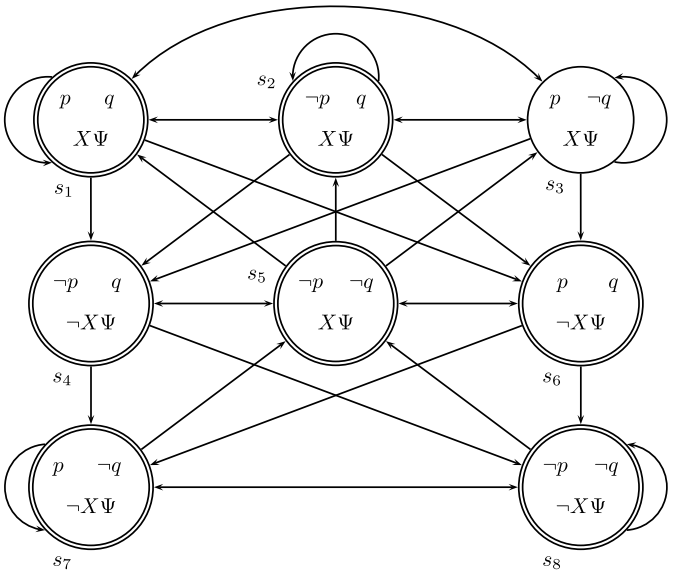


Fig. 6 Tableau for LTL formula $(p \mathbf{U} q)$

Example 5 We conclude this section by illustrating the tableau construction on an LTL formula $\Psi = p \mathbf{U} q$. The set of elementary formulas of Ψ is $el(\Psi) = \{X\Psi, p, q\}$. Thus, the tableau has three atomic propositions, one for each elementary formula. The transition relation R_T is defined as follows:

$$R_T = (X\Psi) \iff q' \vee \left(p' \wedge (X\Psi)'\right)$$

That is, any state in which the atomic proposition $X\Psi$ is true has a transition to every state in which either q is true, or both p and $X\Psi$ are true. Finally, there is a single fairness constraint $\mathscr{F}_\varphi = \{(p \wedge X\Psi) \implies q\}$.

The tableau is shown in Fig. 6. Every fair computation of the tableau starts at either s_1, s_2, s_3, s_4, or s_6 and passes infinitely often through one of the doubly bordered fair states. Note that state s_3 is the only unfair one. Any execution of the tableau that is trapped in state s_3 corresponds to a computation π on which p is always true and q is always false. Clearly, π violates the property $(p \mathbf{U} q)$, and the fairness constraint excludes π from legal executions of the tableau. □

8.5 Push-Down Symbolic Model Checking

In this section, we present a symbolic BDD-based algorithm for model-checking reachability (i.e., safety) properties of push-down systems (PDS). This presentation is focused on the BDD aspect of the algorithm. More details on PDSs and various model-checking algorithms for them are available in Chap. 17. We assume that the

PDS is specified as a Boolean program *BP*, and that the reachability property is specified by an *ERROR* location in *BP*. For simplicity, we assume that the Boolean program is defined by a graph whose nodes correspond to control-flow locations, and whose edges correspond to statements. Let V be a set of Boolean variables, and V' and V'' be, respectively, the sets of their primed and double-primed versions. Given a set of variables $X \subseteq V$, we write $id(X)$ to mean the formula $\bigwedge_{v \in X} v \iff v'$. For simplicity, we often write $id(x_1, \ldots, x_n)$ to mean $id(\{x_1, \ldots, x_n\})$.

Definition 5 (Variable substitution) Given a BDD b, we write $b[V/V']$ to mean the BDD obtained from b by replacing simultaneously each $v \in V$ with its corresponding primed version $v' \in V'$. We define BDDs $b[V/V'']$, $b[V'/V'']$, etc. analogously. We write $b[V/V'][V'/V'']$ to mean the BDD $(b[V/V'])[V'/V'']$, and so on.

Definition 6 (Boolean program) A Boolean program is a five-tuple $(Loc, Init, GV, LV, E)$ where

- Loc is a set of control-flow locations;
- $Init \in Loc$ is the initial control-flow location;
- $GV \subseteq V$ is a set of global Boolean variables;
- $LV : Loc \mapsto \mathscr{P}(V)$ maps each control-flow location l to the set of local Boolean variables at l; we write $LV'(l)$ to mean the set of primed versions of the variables in $LV(l)$.
- $E = (NE, CE, FE)$ is a triple denoting three types of control-flow edges:
  - $NE \subseteq Loc \times Bdd \times Loc$ is the set of normal edges. If $e = (l, R, l')$ is a normal edge, then R is a BDD over $GV \cup LV(l) \cup GV' \cup LV'(l')$ that relates the values of variables before and after the execution of e.
  - $CE \subseteq Loc \times Loc \times Loc$ is the set of call edges: a call edge $e = (l_c, l_e, l_r)$ denotes a function call where l_c is the call-site, l_e is the entry location of the called function, l_r is the caller location where the control returns after the call terminates.
  - $FE \subseteq Loc \times Loc$ is the set of function edges: a function edge $e = (l_{in}, l_{out})$ denotes a function body with l_{in} and l_{out} as its entry and exit locations, respectively.

A *state* of a Boolean program at location l is a valuation of all global variables and all local variables at l.

Example 6 Consider the Boolean program defined by the pseudo-code in Fig. 7 and shown graphically in Fig. 8. In our formalism, the Boolean program is given by the five-tuple $(Loc, Init, GV, LV, E)$ where

- $Loc = \{1, 2, \ldots, 9\}$, $Init = 1$, $GV = \{p, r\}$
- LV maps locations $1, 2, \ldots, 7$ to $\{v\}$ and $8, 9$ to $\emptyset$.
- Each normal edge $(l, r, l') \in NE$ is shown in Fig. 8 by an unbroken arrow from l to l' labeled by r.

```
bool p                                  ▷ global variable used to pass parameters to functions
bool r                                  ▷ global variable used to return values from functions
function MAIN                           ▷ top-level function
    bool v                              ▷ local variable
    1: p := v                           ▷ entry location
    2: NEG()                            ▷ first function call
    3: p := r
    4: NEG()                            ▷ second function call
    5: if r ≠ v then goto 7
    6: return                           ▷ exit location
    7: goto 6                           ▷ error location

function NEG
    8: r :=!p                           ▷ entry location—set return value to negation of argument
    9: return                           ▷ exit location
```

Fig. 7 Pseudo-code of an example Boolean program

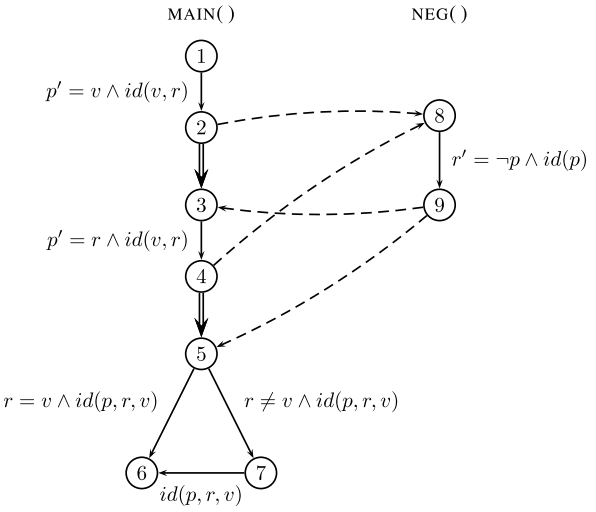


Fig. 8 Control locations and edges of the Boolean program in Fig. 7

- The set of call edges is

$$CE = \{ \underbrace{(2, 8, 3)}_{\text{first call to NEG}} , \underbrace{(4, 8, 5)}_{\text{second call to NEG}} \}.$$

- The set of function edges is

$$FE = \{\underbrace{(1, 6)}_{\text{MAIN}}, \underbrace{(8, 9)}_{\text{NEG}}\}.$$

We assume, without loss of generality, that global variables are disjoint from local variables. We now present a symbolic BDD-based algorithm CHECKBPSAFETY whose inputs are

```
1: var ρ                                          ▷ map from locations to reachable relations
2: var Σ                                          ▷ map from call locations to function summaries
3: function CHECKBPSAFETY(BP, S_0, Err)           ▷ BP = (Loc, Init, GV, LV, E), Err ∈ Loc
4:     COMPUTEREACH(BP, S_0)                      ▷ compute reachable states at each location
5:     return ρ[Err] = 0

6: function IDREL(X)                              ▷ X ⊆ V
7:     return Bdd(id(X))                          ▷ identity relation over variables in X

8: function UPDATESTATE(l, s)
9:     s' := s ∧ ¬ρ(l)
10:    if s' ≠ 0 then                             ▷ if some new states have been reached
11:        W[l] := W[l] ∨ s'  ;  ρ[l] := ρ[l] ∨ s'   ▷ update reachable relation and worklist

12: function COMPUTEREACH(BP, S_0)                ▷ BP = (Loc, Init, GV, LV, (NE, CE, FE))
13:    ρ[Init] := S_0 ∧ IDREL(GV ∪ LV(Init))      ▷ initialize reachable relation at Init
14:    ∀l ∈ Loc \ {Init} . ρ[l] := 0              ▷ initialize other reachable relations
15:    ∀(l_in, l_out) ∈ FE . Σ[l_in] := 0         ▷ initialize summaries
16:    W[Init] := ρ[Init]                         ▷ initialize worklist
17:    while ∃l . W[l] ≠ 0 do
18:        s := W[l]  ;  W[l] := 0
19:        for all (l, R, l') ∈ NE do             ▷ process normal edges
20:            X := R[V'/V''][V/V']
21:            s' := (∃V' . s ∧ X)[V''/V']        ▷ post over transition relation
22:            UPDATESTATE(l', s')                ▷ update next location
23:        for all (l, l_e, l_r) ∈ CE do          ▷ process call edges starting at l
24:            s_e := (∃V . ∃LV'(l) . s)[V'/V] ∧ IDREL(GV ∪ LV(l_e))
25:            UPDATESTATE(l_e, s_e)              ▷ update initial location of called function
26:            X := (Σ(l_e) ∧ IDREL(LV(l)))[V'/V''][V/V']
27:            s_r := (∃V' . s ∧ X)[V''/V']       ▷ post over current summary
28:            UPDATESTATE(l_r, s_r)              ▷ update location after call returns
29:        for all (l_in, l) ∈ FE do              ▷ process call edges returning from l
30:            X := s ∧ ¬Σ(l_in)
31:            if X ≠ 0 then                      ▷ if summary needs to be updated
32:                Σ(l_in) := Σ(l_in) ∨ X         ▷ update summary
33:                for all (l_c, l_in, l_r) ∈ CE do   ▷ process each function call
34:                    X_r := (X ∧ IDREL(LV(l)))[V'/V''][V/V']
35:                    s_r := (∃V' . ρ[l_c] ∧ X_r)[V''/V']   ▷ post over new summary
36:                    UPDATESTATE(l_r, s_r)      ▷ update location after call returns
37:    return
```

Fig. 9 Pseudo-code for symbolic model checking of a Boolean program

1. a Boolean program $BP = (Loc, Init, GV, LV, E)$;
2. an initial state S_0 represented by a BDD $\mathbf{S_0}$ over $GV \cup LV(Init)$;
3. an *ERROR* location $Err \in Loc$.

The algorithm outputs TRUE if and only if there does not exist an execution of BP starting from the initial state S_0 that reaches Err. The pseudo-code for the algorithm is given in Fig. 9. The algorithm iteratively computes two main data structures:

1. A map ρ from control-flow locations to "reachable relations". Given a location l we write $\mathscr{V}(l) = \mathscr{P}(GV \cup LV(l))$ to mean the set of all states at l. Let l be a control location in BP in a function f. Let the entry location of f be l_0. Informally, $\rho[l]$ consists of all pairs of states $(s, s') \in \mathscr{V}(l_0) \times \mathscr{V}(l)$ such that (i) there is an execution of BP from an initial state in S_0 that calls f with state s; and (ii) there is an execution of f that begins at l_0 with state s and reaches l with state s'.
2. A map Σ from entry locations of functions to their "reachable global summaries". Let a global state be a valuation of GV. Let f be a function in BP with entry location l_0. Then $\Sigma(l_0)$ consists of pairs $(s, s') \in \mathscr{P}(GV) \times \mathscr{P}(GV)$ such that (i) there is an execution of BP from an initial state in S_0 that calls f with global state s; and (ii) there is an execution of f that begins at l_0 with global state s and terminates with global state s'. Note that if l_1 is the exit location of f, then $\Sigma(l_0)$ equals the projection of $\rho(l_1)$ over global variables. Indeed our algorithm maintains this invariant by updating $\Sigma(l_0)$ whenever $\rho(l_1)$ changes (see Lines 29–36).

The top-level function is CHECKBPSAFETY. It invokes COMPUTEREACH to compute ρ and Σ, and returns TRUE iff $\rho(Err) = \mathbf{0}$. By the definition of ρ, we know that $\rho(Err) = \mathbf{0}$ if and only if Err is unreachable.

The main function is COMPUTEREACH. It first initializes ρ (Lines 13–14) and Σ (Line 15). It then uses a worklist to iteratively update ρ and σ (Lines 16–37) until a fixed point is reached. It invokes two helper functions: (i) IDREL(X) returns a relation that equates all variables in X with their primed versions; and (ii) UPDATESTATE(l, $\mathbf{s}$) adds $\mathbf{s}$ to the reachable relation at l and updates the worklist if necessary.

The body of the iteration (Lines 18–36) extracts an element from the worklist and processes it. The processing depends on the type of the control location corresponding to the extracted worklist element, and falls into three categories:

- If the location is the source of a normal edge (Lines 19–22), an image is computed and propagated to the successor of the edge.
- If the location is a call-site (Lines 23–28), then two steps are performed: (a) an appropriate image of the current state is propagated to the entry location of the called function (Lines 24–25); and (b) the current summary of the called function is used to compute an image of the current state and the image is propagated to the control-flow location where the function call returns (Lines 26–28).
- If the location is the exit location of a function f (Lines 29–36), then the summary of f is updated (Lines 30–32) and the new summary is used to update the reachable relations at all possible locations where a call to f might return (Lines 33–36).

Example 7 Suppose that CHECKBPSAFETY is called with the Boolean program from Example 6 and with $S_0 = \text{TRUE}$ and $Err = 7$. First, CHECKBPSAFETY calls COMPUTEREACH. During the execution of COMPUTEREACH, ρ, Σ, and W are

initialized as follows:

$$\rho[1] \mapsto id(p, r, v), \quad \forall i \in \{2, \ldots, 9\} \,.\, \rho[i] \mapsto \mathbf{0}$$
$$\Sigma[1] = \Sigma[8] \mapsto \mathbf{0}, \quad W[1] \mapsto id(p, r, v)$$

In the rest of this example, we only mention mappings to BDDs that are not $\mathbf{0}$. Once ρ, Σ, and W are initialized, the main loop (Lines 17–36) is executed. After each iteration, the values of ρ, Σ, and W are updated as follows:

- After iteration 1, we have $\rho[2] = W[2] \mapsto id(r, v) \wedge p' = v$.
- During iteration 2, we start exploring the first call to NEG. At the end of iteration 2 we have $\rho[8] = W[8] \mapsto id(p, r)$.
- After iteration 3, we have $\rho[9] = W[9] \mapsto id(p) \wedge r' = \neg p$.
- During iteration 4, we complete exploring the first call to NEG, update NEG's summary, and return to the first call-site of NEG. At the end of this iteration, we have $\Sigma[8] \mapsto id(p) \wedge r' = \neg p$ and $\rho[3] = W[3] \mapsto id(v) \wedge p' = v \wedge r' = \neg v$.
- After iteration 5, we have $\rho[4] = W[4] \mapsto id(v) \wedge p' = \neg v \wedge r' = \neg v$. The mapping σ is unchanged.
- During iteration 6, we use the existing summary of NEG at the second call to NEG. At the end of this iteration we have $\rho[5] = W[5] \mapsto id(v) \wedge p' = \neg v \wedge r' = v$.
- After iteration 7, we have $\rho[6] = W[6] \mapsto id(v) \wedge p' = \neg v \wedge r' = v$.
- After iteration 8, the worklist W is empty and COMPUTEREACH returns.

After COMPUTEREACH returns, we have $\rho[Err] \mapsto \mathbf{0}$. Therefore, CHECKBPSAFETY returns TRUE, which is the correct result since *Err* is unreachable.

Note that all computations in CHECKBPSAFETY are symbolic and performed using BDDs. We do not discuss the correctness of this algorithm since it is equivalent to the BEBOP [3] algorithm.

8.6 Conclusion

This chapter presents the key ideas and techniques that embody the area of BDD-based symbolic model checking. Starting with hardware model checking, symbolic techniques have helped advance both the theory and practice of formal verification in significant ways. Most state-of-the-art symbolic model checkers today, such as NuSMV [13], employ BDD-based techniques as part of their verification strategies. BDDs have also been implemented in the form of robust libraries, notably CUDD [31], that have been used in a wide variety of applications. At the same time, a number of research directions remain open and under active investigation. We believe that this bodes well for existing and future researchers, and we hope this chapter will at least be a gentle yet accurate introduction to this rich and complex topic.

Acknowledgements This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Department of Defense. This material has been approved for public release and unlimited distribution. DM-0000690

References

1. Akers, S.B.: Binary decision diagrams. IEEE Trans. Comput. **27**(6), 509–516 (1978)
2. Aziz, A., Tasiran, S., Brayton, R.K.: BDD variable ordering for interacting finite state machines. In: Proceedings of the 31st ACM IEEE Design Automation Conference (DAC '94), pp. 283–288. ACM, San Diego (1994)
3. Ball, T., Rajamani, S.K.: Bebop: a path-sensitive interprocedural dataflow engine. In: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '01), pp. 97–103. ACM, Snowbird (2001)
4. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded Model Checking. Academic Press, San Diego (2003)
5. Boute, R.T.: The binary decision machine as programmable controller. Euromicro Newsl. **2**(1), 16–22 (1976). doi:10.1016/0303-1268(76)90033-X
6. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Trans. Comput. **35**(8), 677–691 (1986)
7. Burch, J., Clarke, E.M., Long, D.E., McMillan, K., Dill, D.L.: Symbolic model checking for sequential circuit verification. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. **13**(4), 401–424 (1994)
8. Burch, J., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. In: Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science (LICS '90), pp. 1–33. IEEE, Washington (1990)
9. Cabodi, G., Camurati, P., Quer, S.: Improving symbolic traversals by means of activity profiles. In: Proceedings of the 36th ACM IEEE Design Automation Conference (DAC '99), pp. 306–311. ACM, New Orleans (1999)
10. Cabodi, G., Camurati, P., Quer, S.: Improving the efficiency of BDD-based operators by means of partitioning. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. **18**(5), 545–556 (1999)
11. Cabodi, G., Loiacono, C., Palena, M., Pasini, P., Patti, D., Quer, S., Vendraminetto, D., Biere, A., Heljanko, K.: Hardware model checking competition 2014: an analysis and comparison of model checkers and benchmarks. J. Satisf. Boolean Model. Comput. **9**, 135–172 (2015)
12. Cho, H., Hachtel, G.D., Macii, E., Plessier, B., Somenzi, F.: Algorithms for approximate FSM traversal based on state space decomposition. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. **15**(12), 1465–1478 (1996)
13. Cimatti, A., Clarke, E.M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: an opensource tool for symbolic model checking. In: Brinksma, E., Larsen, K.G. (eds.) Proceedings of the 14th International Conference on Computer Aided Verification (CAV '02). LNCS, vol. 2404, pp. 359–364. Springer, Copenhagen (2002)
14. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Trans. Program. Lang. Syst. **8**(2), 244–263 (1986)
15. Clarke, E.M., Grumberg, O., Hamaguchi, K.: Another look at LTL model checking. Form. Methods Syst. Des. **10**(1), 47–71 (1997)
16. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (2000)
17. Coudert, O., Berthet, C., Madre, J.C.: Verification of synchronous sequential machines based on symbolic execution. In: Sifakis, J. (ed.) Automatic Verification Methods for Finite State Systems. LNCS, vol. 407, pp. 365–373. Springer, Grenoble (1989)

18. Hojati, R., Krishnan, S.C., Brayton, R.K.: Early quantification and partitioned transition relations. In: Proceedings of the 1996 International Conference on Computer Design: VLSI in Computers and Processors (ICCD '96), pp. 12–19. IEEE, Austin (1996)
19. Lee, C.Y.: Representation of switching circuits by binary-decision programs. Bell Labs Tech. J. **38**(4), 985–999 (1959)
20. Lichtenstein, O., Pnueli, A.: Checking that finite-state concurrent programs satisfy their linear specification. In: Proceedings of the 12th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '85), pp. 97–107. ACM, New Orleans (1985)
21. McMillan, K.L.: Interpolants and symbolic model checking. In: Cook, B., Podelski, A. (eds.) Proceedings of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI '07), Springer-Verlag, Nice, France, January 14–16, 2007. LNCS, vol. 4349, pp. 89–90. Springer, New York (2007)
22. Minato, S.: Zero-suppressed BDDs and their applications. Int. J. Softw. Tools Technol. Transf. **3**(2), 156–170 (2001)
23. Moon, I.H., Hachtel, G.D., Somenzi, F.: Border-block triangular form and conjunction schedule in image computation. In: Hunt, W.A. Jr., Johnson, S.D. (eds.) Proceedings of the 3rd International Conference on Formal Methods in Computer-Aided Design (FMCAD '00). LNCS, vol. 1954, pp. 73–90. Springer, Austin (2000)
24. Panda, S., Somenzi, F., Plessier, B.: Symmetry detection and dynamic variable ordering of decision diagrams. In: Proceedings of the 1994 International Conference on Computer-Aided Design (ICCAD '94), pp. 628–631. IEEE, San Jose (1994)
25. Park, D.M.R.: Finiteness is mu-ineffable. Theor. Comput. Sci. **3**(2), 173–181 (1976)
26. Ranjan, R.K., Aziz, A., Brayton, R.K., Plessier, B., Pixley, C.: Efficient BDD algorithms for FSM synthesis and verification. In: Proceedings of the IEEE/ACM International Workshop on Logic Synthesis (IWLS'95), Lake Tahoe, CA (1995)
27. Ravi, K., Somenzi, F.: High-density reachability analysis. In: Proceedings of the 1995 International Conference on Computer-Aided Design (ICCAD '95), pp. 154–158. IEEE, San Jose (1995)
28. Rozier, K.Y.: Linear temporal logic symbolic model checking. Comput. Sci. Rev. **5**(2), 163–203 (2011)
29. Rudell, R.: Dynamic variable ordering for ordered binary decision diagrams. In: Lightner, M.R., Jess, J.A.G. (eds.) Proceedings of the 1993 International Conference on Computer-Aided Design (ICCAD '93), pp. 42–47. IEEE, Santa Clara (1993)
30. Shannon, C.E.: A symbolic analysis of relay and switching circuits. Trans. Am. Inst. Electr. Eng. **57**(12), 713–723 (1938)
31. Somenzi, F.: CUDD: CU decision diagram package. http://vlsi.colorado.edu/~fabio/CUDD/
32. Touati, H.J., Savoj, H., Lin, B., Brayton, R.K., Sangiovanni-Vincentelli, A.L.: Implicit state enumeration of finite state machines using BDDs. In: Proceedings of the 1990 International Conference on Computer-Aided Design (ICCAD '90), pp. 130–133. IEEE, Santa Clara (1990)
33. Xu, J., Williams, M., Mony, H., Baumgartner, J.: Enhanced reachability analysis via automated dynamic netlist-based hint generation. In: Proceedings of the 12th International Conference on Formal Methods in Computer-Aided Design (FMCAD '12), pp. 157–164. IEEE, Cambridge (2012)

Chapter 9
Propositional SAT Solving

Joao Marques-Silva and Sharad Malik

Abstract The Boolean Satisfiability Problem (SAT) is well known in computational complexity, representing the first decision problem to be proved NP-complete. SAT is also often the subject of work in proof complexity. Besides its theoretical interest, SAT finds a wide range of practical applications. Moreover, SAT solvers have been the subject of remarkable efficiency improvements since the mid-1990s, motivating their widespread use in many practical applications including Bounded and Unbounded Model Checking. The success of SAT solvers has also motivated the development of algorithms for natural extensions of SAT, including Quantified Boolean Formulas (QBF), Pseudo-Boolean constraints (PB), Maximum Satisfiability (MaxSAT) and Satisfiability Modulo Theories (SMT) which also see application in the model-checking context. This chapter first covers the organization of modern conflict-driven clause learning (CDCL) SAT solvers, which are used in the vast majority of practical applications of SAT. It then reviews the techniques shown to be effective in modern SAT solvers.

9.1 Introduction

Given a propositional logic formula, determining whether there exists a variable assignment such that the formula evaluates to true is referred to as the Boolean Satisfiability Problem, commonly abbreviated as SAT. SAT has seen much theoretical interest as the canonical NP-complete problem [30]. Given its NP-Completeness, it is very unlikely that there exists any polynomial algorithm for SAT. However, NP-Completeness does not exclude the possibility of finding algorithms that are efficient enough to solve many interesting SAT instances. In addition to model checking, the subject of this book, these instances arise from many diverse areas—many practical problems in AI planning [61], circuit testing [107], and software modeling [54]

J. Marques-Silva (✉)
University of Lisbon, Lisbon, Portugal
e-mail: jpms@ciencias.ulisboa.pt

S. Malik
Princeton University, Princeton, NJ, USA

E.M. Clarke et al. (eds.), *Handbook of Model Checking*,
DOI 10.1007/978-3-319-10575-8_9

can be formulated as SAT instances. This has motivated research in practically efficient SAT solvers. This research has resulted in the development of several SAT algorithms that have seen practical success. These algorithms are based on various principles such as resolution [33], search [32], local search and random walk [98], Binary Decision Diagrams [25], Stålmarck's algorithm [100], and others. Some of these algorithms are complete, while others are stochastic methods. For a given SAT instance, complete SAT solvers can either find a solution (i.e., a satisfying variable assignment) or prove that no solution exists. Stochastic methods, on the other hand, cannot prove the instance to be unsatisfiable even though they may be able to find a solution for certain kinds of satisfiable instances quickly. Stochastic methods have applications in domains such as AI planning [61] and FPGA routing [87], where instances are likely to be satisfiable and proving unsatisfiability is not required. However, for many other domains, including verification using model checking, the primary task is to prove unsatisfiability of the instances. Applications of SAT to model checking arise in bounded model checking [20], as well as interpolant—[83] and induction—[99] based approaches to unbounded model checking. For these, complete SAT solvers are a requirement.

In recent years search-based algorithms based on the well-known Davis–Logemann–Loveland algorithm [32] (sometimes referred to as the DPLL algorithm for historical reasons) are emerging as some of the most efficient methods for complete SAT solvers. Researchers have been working on DPLL-based SAT solvers for about fifty years. In the last ten years we have seen significant growth and success in SAT solver research based on the DPLL framework. Earlier SAT solvers based on DPLL include Tableau (NTAB) [31], POSIT [40], 2cl [112] and CSAT [36] among others. In the mid-1990s, Marques-Silva and Sakallah in the GRASP SAT solver [80, 81], and Bayardo and Schrag in the relsat SAT solver [14] proposed to augment the original DPLL algorithm with non-chronological backtracking and conflict-driven clause learning (CDCL). These techniques greatly improved the efficiency of the DPLL algorithm for structured (in contrast to randomly generated) SAT instances. Many practical applications emerged (e.g., [20, 54, 87]), which pushed these solvers to their limits and provided strong motivation for finding even more efficient algorithms. This led to a new generation of solvers such as SATO [118], Chaff [86], BerkMin [44] and more recently MiniSAT [38] and PicoSAT [19] which pay a lot of attention to optimizing various aspects of the DPLL algorithm. Some of these deal with efficient implementations of specific steps in the DPLL and CDCL, e.g., unit-propagation in SATO and Chaff, and others with more efficient search space pruning such as the locality-based search in Chaff. The results are some very efficient SAT solvers that can often solve SAT instances generated from industrial applications with tens of thousands or even millions of variables.

A DPLL-based SAT solver is a relatively small piece of software. Many of the solvers mentioned above have only a few thousand lines of code (these solvers are mostly written in C or C++, for efficiency reasons). However, the algorithms involved are quite complex and significant attention is focused on various aspects of the solver such as coding, data structures, choosing algorithms and heuristics for specific steps, and parameter tuning. In this chapter we chart the journey from

the original basic DPLL framework through the introduction of efficient techniques within this framework culminating in state-of-the-art CDCL solvers. Given the depth of literature in this field, it is impossible to do this in any comprehensive way; rather, we focus on techniques with consistently demonstrated efficiency in available solvers. While for the most part we focus on techniques within the basic DPLL search framework, we will also briefly describe other approaches and look at some possible future research directions.

The chapter is organized as follows. Section 9.2 introduces the notation used throughout the chapter. Section 9.3 provides an overview of modern CDCL SAT solvers. Section 9.4 details the key techniques that are used in CDCL SAT solvers. Section 9.5 provides a brief overview of SAT-based problem solving, highlighting a number of problems of interest to model checking. Finally, Sect. 9.6 concludes the chapter.

9.2 Preliminaries

This section introduces the notation used in the remainder of the chapter. Standard propositional logic definitions are used throughout the chapter (e.g., [21, 62]). Boolean formulas are represented in calligraphic font, e.g., $\mathcal{F}, \mathcal{H}, \mathcal{S}, \mathcal{U}, \ldots$ Boolean variables are represented with lowercase letters from the start or the end of the alphabet, e.g., $a, b, c, \ldots, r, s, t, u, v, w, x, y, z$. Whenever necessary, subscripts can be used, e.g., $x_1, w_1, \ldots$ An atom is a Boolean variable. A literal is a variable x or its complement $\neg x$. For notational convenience, the complement of a variable x is represented as $\bar{x}$. A Boolean formula $\mathcal{F}$ is defined inductively over a set of propositional variables, with the standard logical connectives, $\neg$, $\wedge$, $\vee$, as follows:

1. An atom is a Boolean formula.
2. If $\mathcal{F}$ is a Boolean formula, then $(\neg\mathcal{F})$ is a Boolean formula. (When $\mathcal{F}$ represents an atom x, $\neg\mathcal{F}$ is represented by $\bar{x}$.)
3. If $\mathcal{F}$ and $\mathcal{G}$ are Boolean formulas, then $(\mathcal{F} \vee \mathcal{G})$ is a Boolean formula.
4. If $\mathcal{F}$ and $\mathcal{G}$ are Boolean formulas, then $(\mathcal{F} \wedge \mathcal{G})$ is a Boolean formula.

Similar definitions can be developed for the other logic connectives, $\rightarrow$ and $\leftrightarrow$. (The use of parentheses is not enforced, and standard binding rules apply (e.g., [62]), with parentheses being used only to clarify the presentation of formulas.) The variables of a Boolean formula $\mathcal{F}$ are represented by $\mathsf{var}(\mathcal{F})$. Set X is also used to refer to the set of variables of a formula, $X = \mathsf{var}(\mathcal{F})$. A clause c is a non-tautologous disjunction of literals. A term t is a non-contradictory conjunction of literals. Commonly used representations of Boolean formulas include conjunctive and disjunctive normal forms (resp. CNF and DNF). A CNF formula $\mathcal{F}$ is a conjunction of clauses. A DNF formula $\mathcal{F}$ is a disjunction of terms. CNF and DNF formulas can also be viewed as sets of sets of literals. The two representations will be used interchangeably throughout the chapter. In the remainder of the chapter, Boolean formulas are referred to as formulas, which includes CNF formulas and DNF formulas. The necessary qualification will be used when necessary.

Given a formula $\mathcal{F}$, a truth assignment ν is a map from the variables of $\mathcal{F}$ to $\{0, 1\}$, $\nu : \mathsf{var}(\mathcal{F}) \mapsto \{0, 1\}$.

Given a truth assignment ν, the value taken by a formula, denoted $\mathcal{F}^\nu$, is defined inductively as follows:

1. If x is a variable, $x^\nu = \nu(x)$.
2. If $\mathcal{F} = (\neg\mathcal{G})$, then
$$\mathcal{F}^\nu = \begin{cases} 0 & \text{if } \mathcal{G}^\nu = 1 \\ 1 & \text{if } \mathcal{G}^\nu = 0. \end{cases}$$
3. If $\mathcal{F} = (\mathcal{E} \vee \mathcal{G})$, then
$$\mathcal{F}^\nu = \begin{cases} 1 & \text{if } \mathcal{E}^\nu = 1 \text{ or } \mathcal{G}^\nu = 1 \\ 0 & \text{otherwise.} \end{cases}$$
4. If $\mathcal{F} = (\mathcal{E} \wedge \mathcal{G})$, then
$$\mathcal{F}^\nu = \begin{cases} 1 & \text{if } \mathcal{E}^\nu = 1 \text{ and } \mathcal{G}^\nu = 1 \\ 0 & \text{otherwise.} \end{cases}$$

In some contexts, including search algorithms for the Boolean Satisfiability (SAT) problem, a truth assignment is relaxed to be partial, i.e., not all variables are assigned a truth value. A truth assignment is complete if the map is total; otherwise it is partial. For a partial truth assignment, if $\nu(x)$ is not specified, then we write $\nu(x) = u$.

For a CNF formula $\mathcal{F}$, let ν be a truth assignment. A clause c is *satisfied* if there exists a literal $l \in c$, such that $l^\nu = 1$. If all literals of c take value 0, then the clause is *falsified*. If all literals but one are assigned value 0, and the remaining one is unassigned, then the clause is *unit*. Finally, a clause is unresolved if it is neither falsified, nor satisfied, nor unit. A CNF formula is satisfied if all clauses are satisfied, and falsified if at least one clause is falsified.

A truth assignment is *satisfying* for $\mathcal{F}$ (or simply a satisfying truth assignment) if $\mathcal{F}^\nu = 1$. A formula $\mathcal{F}$ is *satisfiable* if it has a satisfying truth assignment; otherwise it is *unsatisfiable*. If a formula $\mathcal{F}$ is satisfiable, we write $\mathcal{F} \not\models \bot$. If a formula $\mathcal{F}$ is unsatisfiable, we write $\mathcal{F} \models \bot$.

Definition 1 (Boolean Satisfiability (SAT)) Given a formula $\mathcal{F}$, the decision problem SAT consists of deciding whether $\mathcal{F}$ is satisfiable.

CDCL SAT solvers, but also DPLL SAT solvers, implement some form of backtracking search. Both CDCL and DPLL SAT solvers branch on variables; these are referred to as *decision variables*.

A key procedure in SAT solvers is the *unit clause rule* [33]: if a clause is unit, then its sole unassigned literal must be assigned value 1 for the clause to be satisfied. The iterated application of the unit clause rule is referred to as *unit propagation* or *Boolean constraint propagation* (BCP) [117]. In modern CDCL solvers, as in most implementations of DPLL, logical consequences are derived with unit propagation.

Unit propagation is applied after each branching step (and also during preprocessing[1]), and is used for identifying variables that must be assigned a specific Boolean value. If a falsified clause is identified, a *conflict* condition is declared, and the algorithm backtracks.

In CDCL SAT solvers, each variable x is characterized by a number of properties, including the *value*, the *antecedent clause* (or just *antecedent*) and the *decision level*, denoted respectively by $\nu(x) \in \{0, u, 1\}$, $\alpha(x) \in \mathcal{F} \cup \{\text{NIL}\}$, and $\delta(x) \in \{-1, 0, 1, \ldots, |X|\}$. A variable x that is assigned a value as the result of applying the unit clause rule is said to be *implied*. The unit clause c used for implying variable x is said to be the antecedent of x, $\alpha(x) = c$. For variables that are decision variables or are unassigned, the antecedent is NIL. Hence, antecedents are only defined for variables whose value is implied by other assignments. The decision level of a variable x denotes the depth of the decision tree at which the variable is assigned a value in $\{0, 1\}$. The decision level for an unassigned variable x is -1, $\delta(x) = -1$. The decision level associated with variables used for branching steps (i.e., *decision assignments*) is specified by the search process, and denotes the current depth of the *decision stack*. The decision stack represents the sequence of branched-upon variables. Hence, a variable x associated with a decision assignment is characterized by having $\alpha(x) = \text{NIL}$ and $\delta(x) > 0$. When describing and analyzing SAT solvers, *implication graphs* [80, 81] are used to graphically depict the application of unit propagation at each decision level, as a consequence of each branching decision. Each node in the implication graph shows a literal, with the incoming edges to each literal identifying the antecedent of the assignment. If a falsified clause is identified by unit propagation, this is marked in the implication graph with a special node $\bot$. The implication graph can be viewed as a graphical representation of the relationship between implied variables and their antecendents.

Figure 1 exemplifies the implication graphs considered in this chapter. This example also illustrates the above definitions. With the exception of decision level 0, a decision literal is associated with each decision level. For example, for decision level 1, the decision literal is w, denoting that w is assigned value 1. For simplicity all examples shown just use positive literals (i.e., variables are always decided or implied value 1). Given the implication graph, the antecedent of a given implied assignment can be inferred from the incoming edges. For example, b is assigned value 1 because a and x are assigned value 1. Hence, the antecedent of b is $(\bar{x} \vee \bar{a} \vee b)$.

A standard operation associated with Boolean formulas is *resolution* [33, 94]. Given clauses $C_1 = (x \vee A)$ and $C_2 = (\bar{x} \vee B)$, where A and B are disjunctions of literals without complemented literals, the resolution of C_1 and C_2 is $C_3 = (A \vee B)$. As shown in Sect. 9.4, resolution serves to explain a wide range of techniques used in modern SAT solvers, including CDCL SAT solvers. For example, unit propagation can be explained with resolution operations and, as illustrated in Sect. 9.4.1, clause learning can also be explained as a sequence of resolution operations. Moreover, resolution is also associated with a number of complete proof systems for SAT (e.g., [62, 111]).

[1] Preprocessing serves to simplify Boolean formulas and is briefly covered in Sect. 9.4.7.

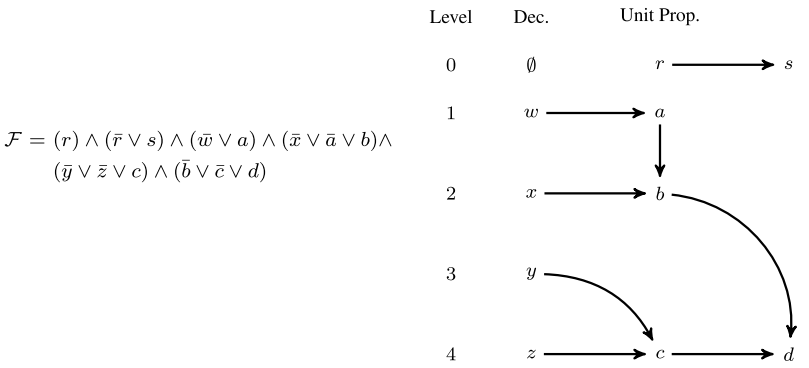


Fig. 1 Example of notation and unit propagation

Modern SAT solvers typically accept CNF formulas [78]. This is due to the inexpensive deduction provided by unit propagation. Procedures for CNF-encoding (or clausifying) arbitrary Boolean formulas are well-known (e.g., [90, 110]).

9.3 CDCL SAT Solvers: Organization

This section provides a high-level description of modern CDCL SAT solvers. Afterwards, Sect. 9.4 details the most important algorithmic techniques associated with CDCL SAT solvers, namely conflict-driven clause learning [80, 81], unique implication points [80, 81], learned clause minimization [105], lazy data structures [86], search restarts [11, 45] and lightweight branching heuristics [86].

Algorithm 1 shows the standard organization of a CDCL SAT solver, which essentially follows the organization of DPLL. With respect to DPLL, the main differences are the call to function ConflictAnalysis each time a conflict is identified, and the call to Backtrack when backtracking takes place. Moreover, the Backtrack procedure allows for backtracking non-chronologically.

In addition to the main CDCL function, the following auxiliary functions are used:

- UnitPropagation consists of the iterated application of the unit clause rule. If a falsified clause is identified, then a conflict indication is returned.
- PickBranchingVariable consists of selecting a variable and assigning it a value.
- ConflictAnalysis consists of analyzing the most recent conflict and learning a new clause from the conflict. The organization of this procedure is described in Sect. 9.4.1.
- Backtrack backtracks to the decision level computed by ConflictAnalysis.
- AllVariablesAssigned tests whether all variables have been assigned, in which case the algorithm terminates indicating that the CNF formula is satisfiable.

Algorithm 1 Typical CDCL algorithm

```
CDCL(F, ν)
 1  if (UnitPropagation(F, ν) == CONFLICT)
 2     then return UNSAT
 3  dl ← 0                                   ▷ Decision level
 4  while (not AllVariablesAssigned(F, ν))
 5        do (x, υ) = PickBranchingVariable(F, ν)
 6           dl ← dl + 1                    ▷ Increment decision level due to new decision
 7           ν ← ν ∪ {(x, υ)}
 8           if (UnitPropagation(F, ν) == CONFLICT)
 9             then β = ConflictAnalysis(F, ν)
10                  if (β < 0)
11                    then return UNSAT
12                    else Backtrack(F, ν, β)
13                         dl ← β            ▷ Decrement decision level due to
                                               backtracking
14  return SAT
```

An alternative criterion to stop execution of the algorithm is to check whether all clauses are satisfied. However, in modern SAT solvers that use lazy data structures, clause state cannot be maintained accurately, and so the termination criterion must be whether all variables are assigned. Thus, in this case the algorithm provides a complete assignment.

Arguments to the auxiliary functions are assumed to be passed by reference. Hence, $\mathcal{F}$ and ν are supposed to be modified during execution of the auxiliary functions.

The typical CDCL algorithm shown does not account for a few often-used techniques, namely search restarts [11, 45] and implementation of clause deletion policies [44]. Search restarts cause the algorithm to restart itself. However, past search history is not erased, for example previously learnt clauses are kept. Clause deletion policies are used to decide learned clauses that can be deleted based on their expected future utility. Clause deletion allows the memory usage of the SAT solver to be kept under control.

9.4 CDCL SAT Solvers

This section reviews the techniques that are common to CDCL SAT solvers. These techniques can be organized as follows:

1. Conflict-driven clause learning [80, 81].
2. Unique implication points [80, 81].
3. Learned clause minimization [105].

Algorithm 2 Main steps of conflict analysis procedure

CONFLICTANALYSIS($\mathcal{F}$, ν)

1 Start at node $\bot$
2 Recursively visit literals of antecedents assigned at current decision level
3 Record complement of antecendent literals assigned at lower decision levels
4 Record complement of branching literal
5 Create clause with recorded literals
6 **return** largest decision level of recorded literals other than the current level

4. Lazy data structures [86].
5. Search restarts [11, 45].
6. Lightweight branching heuristics [86].
7. Additional techniques [7, 44, 89].

9.4.1 Clause Learning and Non-chronological Backtracking

Learning from conflicts has been extensively studied in a number of areas since the 1970s (e.g., [106]). In some contexts, learning from conflicts was shown to be ineffective, both in theory and in practice [9, 115]. Clause learning in SAT solvers [80, 81] is inspired by this earlier work on learning from conflicts, but exhibits important differences. The most important aspect is that clause learning exploits the sequence of unit propagation steps that produces the conflict. In addition, clause learning in SAT solvers exploits UIPs (see Sect. 9.4.2). The original ideas of clause learning in SAT solvers were proposed in the GRASP SAT solver [72, 80, 81]. A recent alternative formalization of clause learning can be found in [78]. This section overviews clause learning by summarizing the main steps and illustrating how these are applied to a simple example.

As the CDCL algorithm is executed, if a falsified clause is identified, conflict analysis is used to create a clause that explains and prevents the same conflict from re-occurring. Algorithm 2 summarizes the main steps of the conflict analysis (and learning) procedure. The input arguments are the CNF formula, and the current set of assignments. Literals implied at the current decision level are traversed, starting from the $\bot$ vertex (which represents the falsified clause). For each traversed literal, the literals in the antecedent are analyzed. A literal assigned at a decision level lower than the current one has its complemented literal recorded, whereas a literal assigned at the current decision level is scheduled to be traversed. The process is repeated until the branching variable for the current decision level is visited.

Figure 2 shows a simple example of unit propagation yielding a conflict. The implication graph summarizes how unit propagation produces the conflict. Algorithm 2 is executed on the implication graph, starting from node $\bot$. Literals a, b, and z are visited, since all are assigned at decision level 3. The recorded literals are $\bar{x}$ and $\bar{z}$. Thus, the created clause is $(\bar{x} \vee \bar{z})$. These steps are shown in Fig. 3.

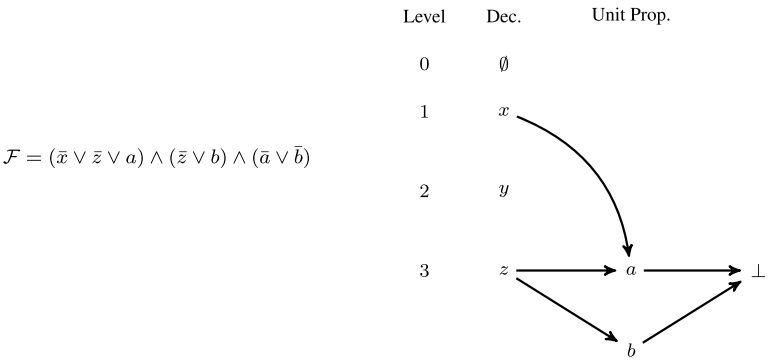


Fig. 2 Clause learning: (a) example formula and (b) conflict after unit propagation

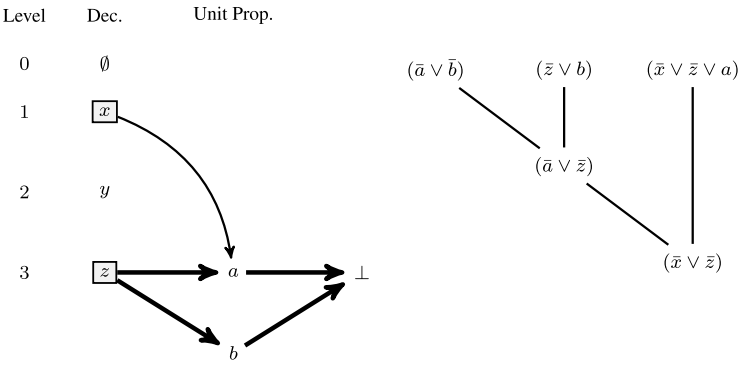


Fig. 3 Clause learning: creating a new clause

Traversed edges are marked with thick lines. Each literal for which the complement is recorded is highlighted and shown inside a box. Moreover, the derivation of the learned clause is formally explained by the application of a sequence of (selected) resolution steps. Hence, clause learning can be viewed as a way to decide which clauses to learn by selective resolution steps. Figure 4 also shows the result after backtracking. The backtrack step shown is the one proposed in [86], which differs somewhat from the backtrack step originally associated with clause learning in the GRASP SAT solver [80, 81]. The GRASP SAT solver delayed backtracking until both assignments had been considered for the branching variable. This would avoid possibly unnecessary (and, in the case of GRASP, expensive) backtracking.

A number of researchers have investigated ways to improve the basic clause learning procedure outlined above (e.g., [6]). Nevertheless, most state-of-the-art SAT solvers implement the basic clause learning procedure, first proposed in GRASP [80, 81], with the backtracking step used in Chaff, but improved with learned clause minimization (which is described in Sect. 9.4.3). Recent work ad-

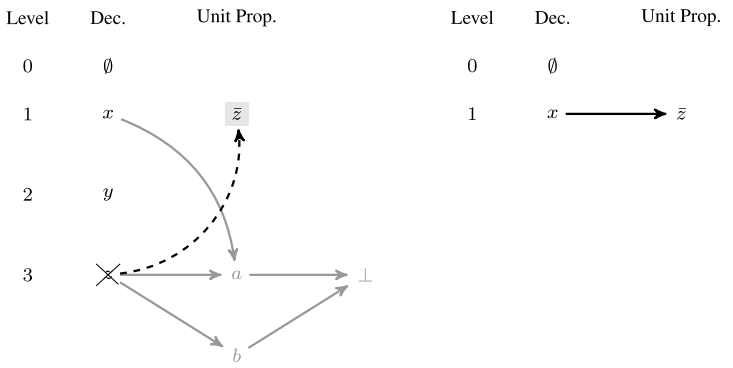


Fig. 4 Clause learning: after backtracking

dresses techniques to decide which clauses are expected to be of interest for the subsequent search [7].

9.4.2 Unique Implication Points

A key aspect of clause learning in SAT solvers is *Unique Implication Points* (UIPs). If unit propagation due to a branching decision yields a conflict, then any dominator [109] of the conflict node with respect to the branching decision is a UIP [80, 81]. UIPs can be related with *failure-driven assertions* [79], used in the context of circuit testing, and mimic, at the logic level, the notion of unique sensitization points (USPs) also used in testing [42]. UIPs serve a number of purposes in CDCL SAT solvers. First, UIPs allow learning of smaller clauses. Second, UIPs allow learning of multiple clauses. The clause learning procedure outlined in Algorithm 2 can be modified to stop when the first dominator is identified. The intuitive justification for this is that assigning the literal associated with the UIP suffices to reproduce the conflict. Hence, the clause learning procedure can terminate by recording the complement of the UIP literal.

Figure 5 illustrates the use of UIPs in clause learning. For this example, without the identification of UIPs, the learned clause would be $(\bar{w} \vee \bar{x} \vee \bar{y} \vee \bar{z})$. This is shown in Fig. 6, where the clause is learned following the steps outlined earlier. However, if clause learning stops at the first UIP, then the learned clause becomes $(\bar{w} \vee \bar{x} \vee \bar{a})$. Observe that stopping at the first UIP essentially consists of performing fewer resolution steps, i.e., the clause learned by stopping at the first UIP is already present in the resolution steps used to derive the learned clause without stopping at the first UIP.

Moreover, observe that, for this concrete example, the learned clause is not only smaller, but induces backtracking to a lower decision level. A straightforward observation is that clauses learned by stopping learning at the first UIP result in backtracking decision levels that are no larger than the decision levels of clauses learned

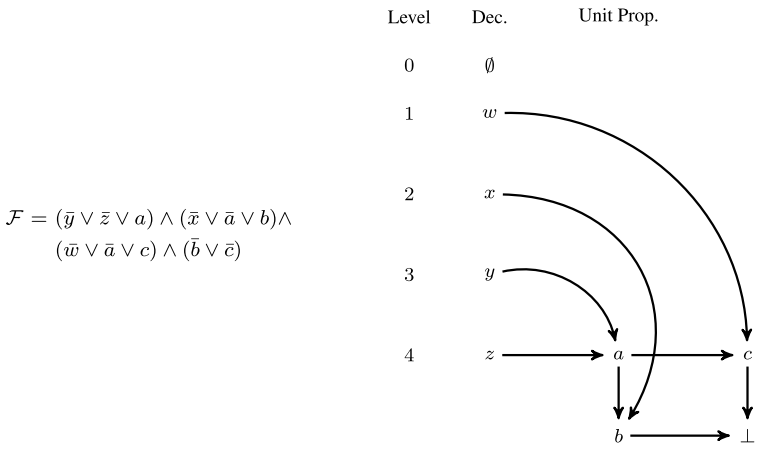


Fig. 5 Unique implication points: (a) example formula and (b) conflict after unit propagation

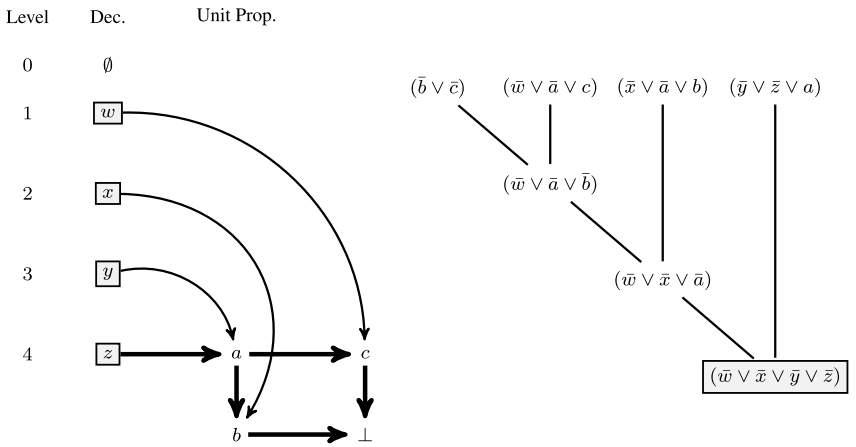


Fig. 6 Clause learning without UIPs

by stopping at the decision literal. A slightly more detailed characterization of this property can be found in [6].

Although the modern usage of UIPs is based on stopping clause learning at the first UIP, the original approach was to learn clauses at every UIP [80, 81]. Recent results, obtained on problem instances from the SAT competitions, suggest that learning clauses at multiple UIPs can improve SAT solver performance [96]. An example of clause learning at multiple UIPs is shown in Fig. 8. As shown in Fig. 9a, conflict analysis by stopping at the first UIP produces the learned clause $(\bar{w} \vee \bar{y} \vee \bar{a})$. However, it is possible to continue learning clauses at each additional UIP. For the example in Fig. 8, z is also a UIP (it is actually the UIP corresponding to the decision variable). Observe that ($x = 1$ and) $z = 1$ implies $a = 1$, and so $a = 0$ implies $z = 0$.

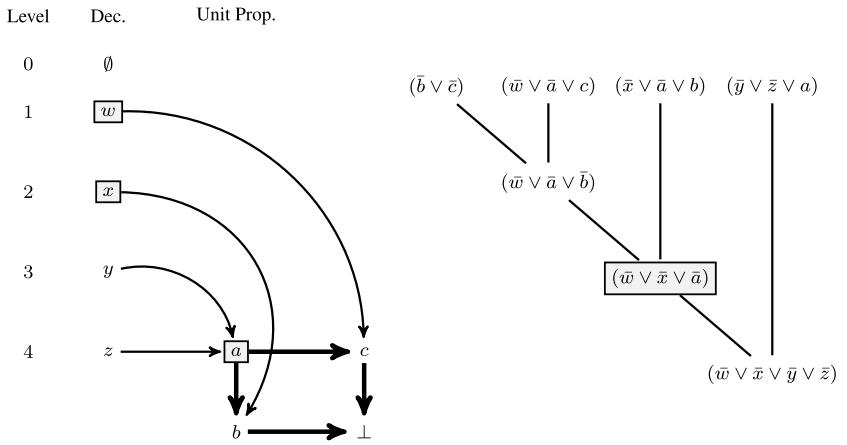


Fig. 7 Clause learning with UIPs

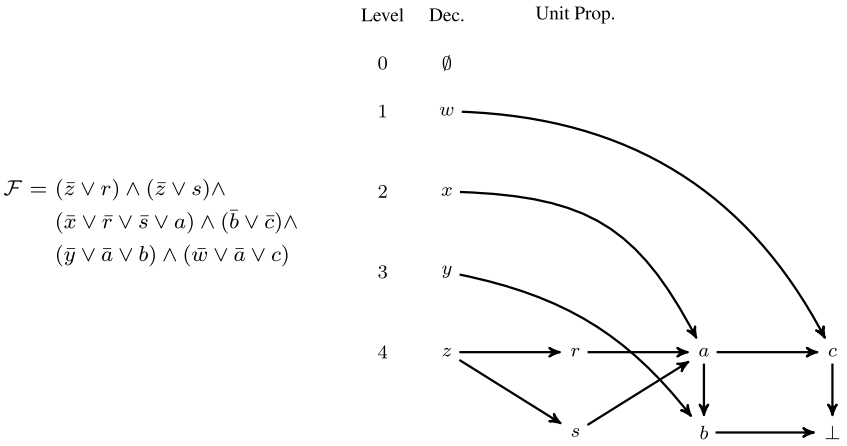


Fig. 8 Multiple UIPs: (a) example formula and (b) conflict after unit propagation

However, this information is not obtained by unit propagation, i.e., $a = 0$ does *not* lead to $z = 0$. Nevertheless, by noting that z is a UIP, the following clause is learned: $(\bar{z} \vee \bar{x} \vee a)$. This is illustrated in Fig. 9b. With this additional clause added to the formula, $a = 0$ now implies $z = 0$ whenever $x = 1$. The clauses obtained by clause learning at multiple UIPs are inspired by, but generalize, the concept of global implications first studied in the area of circuit testing [97].

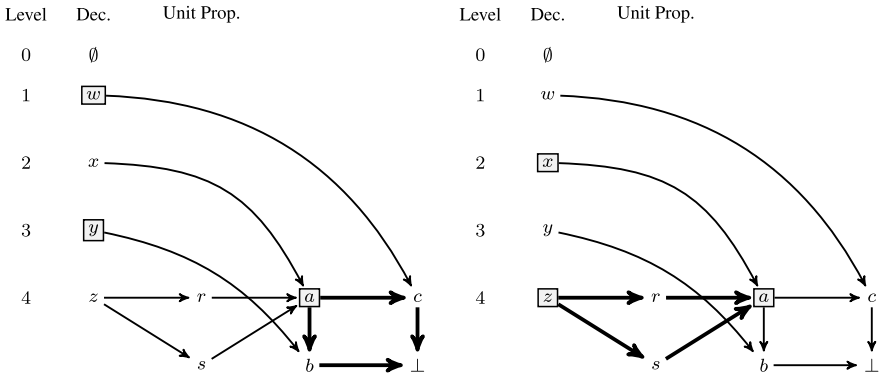


Fig. 9 Multiple UIPs: (a) first UIP clause; and (b) second UIP clause

$$\mathcal{F} = (\bar{x} \vee b) \wedge (\bar{z} \vee \bar{b} \vee c) \wedge (\bar{x} \vee \bar{y} \vee \bar{z} \vee a) \wedge (\bar{a} \vee \bar{c})$$

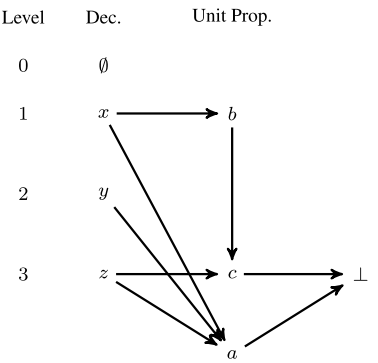


Fig. 10 Learned clause minimization: (a) example formula; (b) conflict after unit propagation

9.4.3 Learned Clause Minimization

The basic clause learning procedure has not changed significantly since the mid-1990s [80, 81]. However, recent SAT solvers exploit a key optimization step after clause learning: *learned clause minimization* [105]. In the mid-2000s, researchers noticed that learned clauses exhibit important redundancies, and that these can be removed with simple procedures. The performance gains obtained with learned clause minimization justify the inclusion of this technique in most modern SAT solvers.

Let $C_1 = (x \vee A)$ and $C_2 = (\bar{x} \vee A \vee B)$ be clauses of $\mathcal{F}$, where A and B are disjunctions of literals. Resolution between C_1 and C_2 produces the clause $C_3 = (A \vee B)$ which subsumes C_2. If C_3 is added to $\mathcal{F}$, then C_2 can be removed from $\mathcal{F}$, since it is subsumed by C_3. This form of resolution is called *self-subsuming resolution* [105]. One clause minimization procedure consists of the iterative application of self-subsuming resolution between a learned clause c and the antecedents of the literals in c [105]. Figure 10 shows an example of clause minimization by self-subsuming resolution. Clause learning without clause minimization, shown in

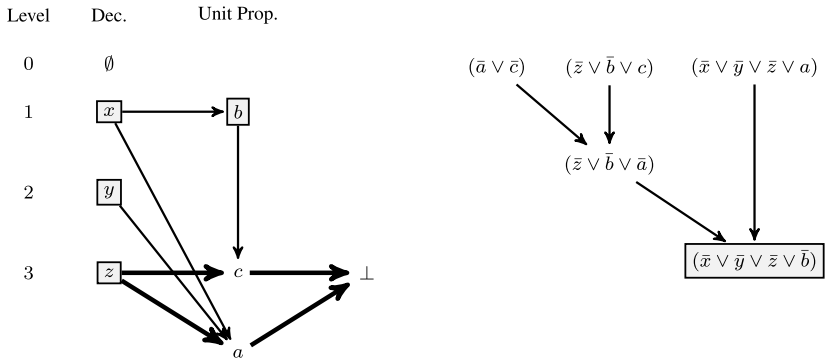


Fig. 11 Clause learning without minimization

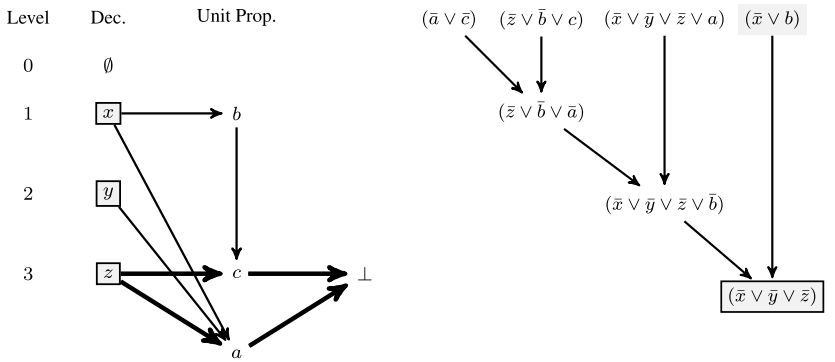


Fig. 12 Minimization with self-subsuming resolution

Fig. 11, yields the learned clause $C_l = (\bar{x} \vee \bar{y} \vee \bar{z} \vee \bar{b})$. However, clause learning followed by self-subsuming resolution between C_l and the antecedent of b yields the clause $C'_l = (\bar{x} \vee \bar{y} \vee \bar{z})$, as shown in Fig. 12. Observe that, in contrast with UIPs, self-subsuming resolution steps are resolution steps which are appended to the resolution derivation to generate the final minimized learned clause.

In practice, self-subsuming resolution is often not enough to effectively minimize learned clauses. An alternative is the so-called *recursive minimization* procedure [105], which is summarized in Algorithm 3. Figure 13 shows an example of applying recursive clause minimization. As shown in Fig. 14, clause learning without minimization yields clause $(\bar{w} \vee \bar{x} \vee \bar{c})$. For this example, self-subsuming resolution cannot be applied, because resolution operations make the resulting clause larger. However, the recursive clause minimization procedure can be used to prove that literal $\bar{c}$ can be dropped from the clause. As shown in Fig. 14b, the traversal from vertex c solely reaches marked vertex w. Hence, the literal $\bar{c}$ can be dropped from the learned clause, and so the final clause becomes $(\bar{w} \vee \bar{x})$.

Algorithm 3 Main steps of recursive clause minimization procedure

RECURSIVECLAUSEMINIMIZATION(c)

```
1  Mark literals in c
2  Implied literals in c are flagged as candidates for removal
3  foreach candidate literal l in c
4       do Traverse implication graph starting from antecedent of l
5          Stop at decision literals or marked literals
6          if Non-marked literal visited
7            then Keep literal l in c
8            else  Drop literal l from c
9  return c
```

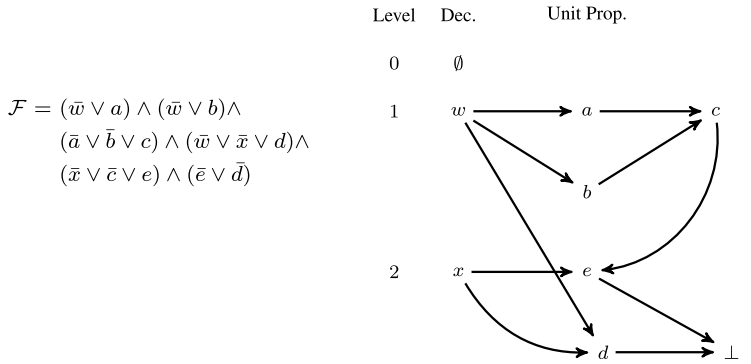


Fig. 13 Learned clause minimization: (a) example formula; (b) conflict after unit propagation

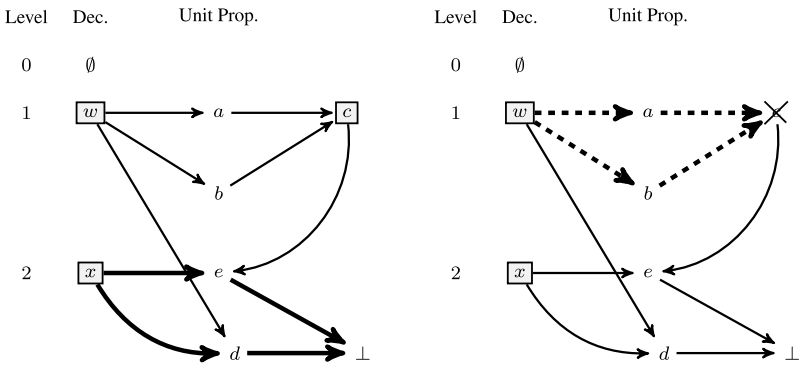


Fig. 14 Clause learning: (a) no minimization; and (b) recursive minimization

9.4.4 Lazy Data Structures

Until the early 2000s most DPLL/CDCL SAT solvers used adjacency lists as the underlying data structure for clause representation [70], with the exception being the

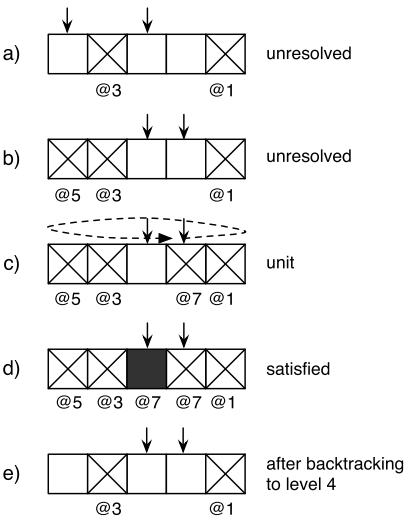


Fig. 15 Operation of watched literals

head-tail representation proposed in SATO [119]. Adjacency lists require L references from literals to clauses, where L denotes the total number of literals. This can become an issue when learning many (possibly large) clauses. The head-tail representation requires between $2 \times C$ and L references from literals to clauses [70, 78], where C denotes the number of clauses. Although more efficient in practice than adjacency lists, the head-tail representation causes overhead when backtracking, besides requiring a varying number of references.

One of the main contributions of the Chaff SAT solver was the use of a new (lazy) data structure, the *watched literals* data structure. The watched literals data structure has several important advantages. First, for each clause only two references from literals to the clause are required. This results in $2 \times C$ references in total. Also, when backtracking, no bookkeeping is required. This provides significant performance gains over the other data structures. As a result, watched literals have become the de facto standard in the implementation of modern SAT solvers (e.g., [19, 38]). Observe that the lowest number of references for each clause is 2, since one must be able to decide when the clause is unit so that unit propagation can be used to assign a value to some variable. Figure 15 illustrates the operation of the watched literals data structure, being adapted from [78]. The example considers a single clause with 5 literals, with the arrows showing the currently watched literals. The current decision level is either 3 or 4, and the clause has 2 literals already assigned value 0 (shown crossed out in the figure). At decision level 5, one of the watched literals is assigned value 0. This requires the algorithm to find another literal to watch, and so the reference is updated. At decision level 7, another watched literal is assigned value 0. In this case, all literals are visited, trying to find a literal that is still unassigned and not watched. In this case none exists, i.e., all literals but one are assigned value 0 and the remaining unassigned literal is already watched. Hence, the clause is declared unit. As a result, the only unassigned literal is assigned value 1 (shown as a black box in the figure), so that the clause becomes satisfied.

Afterwards, the algorithm backtracks to decision level 4. As indicated earlier, there is no need to update the references. Thus, when backtracking, the watched literal data structure requires no bookkeeping.

9.4.5 Search Restarts

Another standard ingredient in modern SAT solvers is search restarts [45]. Research in the late 1990s showed that DPLL SAT solvers exhibit heavy-tail behavior on satisfiable problem instances when the branching heuristic is randomized [45]. This means that the run times of DPLL SAT solvers can exhibit large variations for the same satisfiable instance, and that large run times can happen with non-negligible probability. This observation motivated the proposal of *rapid randomized restarts*, i.e., to restart the search after a fixed (or alternatively increasing) number of conflicts. The increase in the number of conflicts is one possible technique to guarantee that the SAT algorithm is still complete when search restarts are implemented; another is to keep *all* learned clauses [68]. Later work [11] showed that search restarts were also very effective for CDCL SAT solvers, and for solving unsatisfiable problem instances. These conclusions were further substantiated by the implementation of search restarts in the Chaff SAT solver [86].

As with the techniques described in earlier sections, search restarts are commonly used in modern CDCL SAT solvers [19, 38, 78]. In recent years, a number of works have studied different restart policies, including [7, 19, 52, 103].

9.4.6 Lightweight Branching Heuristics

Modern CDCL SAT solvers also exploit so-called lightweight branching heuristics, most notably the VSIDS branching heuristic [86]. The previous generation of branching heuristics [73] maintained counts of assigned literals in each clause. This incurs a significant overhead. For example, in the GRASP SAT solver, branching could account for more than two thirds of the run time [70]. In contrast, lightweight branching heuristics use solely information from conflicts to decide which variables to branch upon. Hence, static or dynamic literal counts are not required. Variables that are involved in more conflicts are more likely to be used for branching than variables not involved in conflicts. This is achieved by associating a metric with each variable, which is incremented for variables involved in conflicts. On average, the most recent conflicts are more relevant than earlier conflicts, since these may no longer be useful for the current state of the search. As a result, VSIDS divides the variable metrics by a constant after a fixed number of conflicts. Besides the low overhead of this heuristic, it also results in what is called locality-based search. Since the variables occurring in recent conflicts are weighted more heavily, the algorithm is biased towards branching on these variables. Thus, the search focuses on

the sub-space of recent conflicts, effectively pruning this sub-space before moving on to other sub-spaces. While only intuitively understood, this has a very significant impact on the size of the search space explored and is credited with the speed-up of this generation of SAT solvers. As with the techniques described in earlier sub-sections, the VSIDS branching heuristic has become a de facto standard in modern CDCL SAT solvers.

9.4.7 Additional Techniques and Recent Trends

This section reviews a few other techniques that are found in modern CDCL SAT solvers. One key issue with CDCL SAT solvers is that the number of learned clauses can become too large. As a result, researchers have developed different solutions for this problem since the mid-1990s. Original solutions were based on restricting the size of learned clauses [14, 80, 81]. More recent work proposes the use of different metrics to decide which clauses to delete [7, 44]. Earlier work considered activity heuristics [44], i.e., if a clause is not used for unit propagation, then it can be marked for deletion. More recent work gives preference to deleting clauses whose literals are distributed by more decision levels [7].

The main change to the organization of branching is the use of *phase saving* [89], i.e., the value of each assigned literal is saved when backtracking takes place. Afterwards, this saved value is reused when that literal is branched upon.

Formula preprocessing has been studied extensively [24, 69, 74]. Recent work has shown that specific forms of preprocessing are effective [37, 57]. Among the many techniques that have been proposed, the most widely used include variable elimination, blocked clause elimination and elimination of subsumed clauses. Moreover, preprocessing techniques have been integrated within SAT solvers, under the general framework of *inprocessing* [58].

Additional promising research directions include algorithm portfolios for SAT [96, 116] and parallel algorithms for SAT [47–49].

9.5 SAT-Based Problem Solving

The importance of SAT solvers is demonstrated by the many problem-solving uses of SAT. This section overviews the different ways in which SAT solvers can be used for solving different problems.

The standard use of SAT solvers is as an engine for solving decision problems, i.e., requiring a yes/no answer. A large number of practical applications of SAT also involve iterative SAT solving, i.e., the problem to be solved requires calling a SAT solver a number of times. Clearly, the number of calls to the SAT solver is paramount in the overall efficiency.

In some cases, the number of calls to the SAT solver is polynomial in the size of the problem instance, but in some other cases the number of calls to the SAT solver is exponential in the worst case.

9.5.1 Incremental SAT

A key issue with iterative use of SAT solvers is how to communicate minimal changes in the formula to the SAT solver and, rather more importantly, how to reuse the learned clauses from previous SAT solver calls. One alternative is to communicate the complete CNF formula each time the SAT solver is to be called. This approach is often referred to as non-incremental, and reuse of learned clauses is not used. Another alternative is to communicate to the SAT solver only the clauses that should be discarded (or deactivated) and the new clauses that should be considered (or activated). This alternative is referred to as incremental, and its use in applications based on iterative SAT solving is now common. The essential ideas for incremental SAT solving are summarized below.

Most modern SAT solvers achieve these goals by using *assumptions* [38]. Clauses in the SAT solver are associated with a new assumption variable. Then, assumption variables are used in each SAT solver call to activate/deactivate clauses. The use of assumptions has important advantages and significant disadvantages. First, any learned clause will keep a record of the clauses explaining its derivation. Thus, activation (resp. deactivation) of assumption variables immediately activates (resp. deactivates) learned clauses that are usable (resp. unusable) in the next SAT solver call.

Another technique to implement incremental SAT, and so to allow reuse of learned clauses, is to use some proof-tracing mechanism [2, 19] (which includes representation of resolution proofs) [19].

Both approaches listed have advantages and disadvantages. Nevertheless, the use of assumptions is more widespread in published work.

9.5.2 Unsatisfiable Cores

In many SAT applications, including model checking, SAT solvers are expected to produce unsatisfiable cores [120], i.e., a subset of the original subformula which was used to prove unsatisfiability. Alternatively, a SAT solver can produce a resolution proof [120]. Unsatisfiable cores find a wide range of applications, including model checking [22], debugging specifications [101], and abstraction refinement [15]. Resolution proofs also find different applications, e.g., in computing interpolants [84].

Two main alternatives exist for computing unsatisfiable cores. The original approach consists of tracing the process of clause learning in CDCL SAT solvers, e.g., by writing an explanation for each learned clause to disk (or keeping it in memory in a separate data structure). Examples of variants of this approach include [2, 19, 120]. A widely used alternative is based on the use of assumption variables (see previous section). When learning clauses, all assumption variables associated with the clauses used for explaining a learned clause are added to the learned clause. Thus, when the SAT solver terminates, instead of producing the empty clause, it produces a clause containing the list of assumption variables of all clauses involved in proving the instance unsatisfiable, i.e., an unsatisfiable core.

9.5.3 CNF Encodings

In most uses of SAT, problems are not initially represented in CNF, e.g., [108]. As a result, a large body of research has been dedicated to encoding richer domains into CNF. Concrete examples include Satisfiability Modulo Theories (SMT), Constraint Satisfaction Problems (CSP), Answer Set Programming (ASP), but also simple extensions of propositional logic, that include non-clausal and pseudo-Boolean (PB) constraints.[2]

Encodings to CNF often address two key aspects. First, the size of the resulting CNF formula, namely whether the size of the encoding is polynomial in the original problem representation. Second, whether the CNF encoding preserves arc-consistency (e.g., see [1, 8, 92]), i.e., whether unit propagation suffices to (i) identify partial assignments that cannot be extended to a satisfying assignment; and (ii) identify any necessary assignments.

A number of ways exist to encode SMT into SAT. An up-to-date review is provided in [63]. Similarly to SMT, there are a number of ways to encode CSP into SAT. An overview of CSP to SAT encodings is provided in [92]. Like with SMT and CSP, there are also different ways to encode ASP into SAT. A recent account is provided in [55].

In many model-checking applications, instances of SAT are naturally non-clausal (e.g., interpolants in interpolant-based model checking [84]). As a result, mechanisms for encoding non-clausal formulas into clausal form have been developed (e.g., [90, 110]). A recent survey of these encodings is provided in [92].

For many practical applications, the domain variables are Boolean and the goal is to encode a pseudo-Boolean (PB) constraint of the general form:

$$\sum_{j=1}^{n} a_j x_j \bowtie b \tag{1}$$

where $\bowtie \in \{<, \leq, =, \geq, >\}$, $a_j \geq 0$, with $j \in \{1, \ldots, n\}$, $b \geq 0$, and x_j are propositional. For analyzing the size of the encodings, a_M denotes the value of the largest coefficient in (1).

A number of special cases of (1) have been extensively studied in the past. These include cardinality constraints of the form AtMost$k$, AtLeast$k$, and Equals$k$:

$$\sum_{j=1}^{n} x_j \bowtie k \tag{2}$$

Of these, constraints of the form AtMost1 have also been extensively studied [92]. (Observe that an AtLeast1 constraint can be trivially encoded with a clause, and so an Equals1 constraint can be encoded with an AtLeast1 and an AtMost1 constraint.)

There is a vast body of work on encoding PB constraints, cardinality constraints and AtMost1/Equals1 constraints [92]. Table 1 shows examples of CNF encodings.

[2] See [21] and references therein.

Table 1 Examples of CNF Encodings

Type	Encoding	# Clauses	Arc-Consistency	Reference
Pseudo-Boolean	Operational	linear	No	[113]
	BDD	exponential	Yes	[39]
	GPWE	$\mathcal{O}(n^3 \log(n) \log(a_{\mathrm{M}}))$	Yes	[8]
	GPWE*	$\mathcal{O}(n^3 \log(a_{\mathrm{M}}))$	Yes	[1]
Cardinality	BDD	$\mathcal{O}(n\,k)$	Yes	[39]
	Seq. Counter	$\mathcal{O}(n\,k)$	Yes	[102]
	Sort. Networks	$\mathcal{O}(n\,\log^2 n)$	Yes	[13, 39]
	Card. Networks	$\mathcal{O}(n\,\log^2 k)$	Yes	[4]
AtMost1	Seq. Counter	$\mathcal{O}(n)$	Yes	[102]
	Bitwise	$\mathcal{O}(n \log n)$	Yes	[41, 91]

9.5.4 Optimization

In many settings, the problem to be solved involves a set of constraints ($\mathcal{F}$) subject to a linear cost function $f = \sum_{x \in X} x$. In Boolean domains, optimization problems can be described as follows:

$$\begin{aligned} &\min \ \textstyle\sum_{i=1}^{n} c_j\, x_j \\ &\text{s.t.} \quad \mathcal{F} \end{aligned} \tag{3}$$

(3) can be solved with algorithms for pseudo-Boolean optimization. For this concrete case, the cost function can be optimized with standard linear or binary search (see, e.g., [95] for an overview).

Alternatively, (3) can be reduced to weighted partial Maximum Satisfiability (e.g., [51]). The original constraints $\mathcal{F}$ are set as hard clauses. Moreover, each term in the cost function can be represented as a soft clause ($\neg x_j$) with cost c_j. A wealth of algorithms have been developed in recent years for MaxSAT. These include branch-and-bound search, iterative SAT solving and (unsatisfiable) core-guided approaches. Recent accounts are provided in [3, 66, 85].

9.5.5 Model Enumeration

In many settings, a SAT solver is required to compute all satisfying assignments. A well-known example is in model checking [59, 83]. Another well-known example is the use of SAT solvers in lazy SMT solvers [12], where satisfying assignments are iteratively computed until a model of the SMT formula is found, or the formula is proved unsatisfiable. An essential step in model enumeration is the identification of prime implicants, e.g., [93].

Given a (total) satisfying assignment for the variables, a prime implicant can be obtained by iteratively checking whether each variable is required for satisfying the

formula [93]. The resulting set of literals is a prime implicant, and its complement can be used for blocking the recomputation of any model that is covered by the prime implicant.

9.5.6 Minimal Sets

A number of applications of SAT solvers involve computing minimal sets. Concrete examples include computing minimal unsatisfiable subsets (MUSes) [16, 46], minimal correction subsets (MCSes) [67, 75], prime implicates (PIs) [23], and minimal models [17, 18], among many others. Recent work shows that all these problems can be solved with the same algorithms [76]. Algorithms for computing minimal sets of Boolean formulas include the following:

- Insertion-based (or constructive) [104].
- Deletion-based (or destructive) [10, 28].
- Dichotomic [50].
- QuickXplain [60].
- Progression [76].

Of these, QuickXplain and Progression offer the best performance in terms of the worst case number of calls to a SAT solver. The deletion-based algorithm is well known, and has been rediscovered in different settings, e.g. [10, 28]. Given a reference set of elements and a monotone predicate P, each element is iteratively removed from the reference set and the predicate is checked on the resulting set. If the predicate holds, the element is dropped from the reference set; otherwise it is kept. In the end, the resulting set is a minimal set.

Depending on the type of minimal set being computed, different approaches exist for reducing the number of calls to a SAT solver. For computing MUSes and PIs, existing techniques include using unsatisfiable cores to remove unnecessary clauses [10, 16, 35] and model rotation [16, 114].

9.5.7 Quantification

Quantified Boolean Formulas (QBF) are Boolean formulas where the variables can either be existentially or universally quantified. Quantification changes the complexity class, and QBF is a well-known PSPACE-Complete problem [26, 62]. In practice, solving QBF formulas turns out to be significantly harder than solving SAT. A large number of approaches have been proposed for deciding QBF formulas, i.e., for deciding whether a formula is true or false. A recent overview of algorithms for QBF is provided in [43, 56].

9.6 Research Directions

Despite a well-defined set of key techniques, CDCL SAT solvers have been the subject of continued improvements over the years. This section outlines possible lines of research in the area of propositional SAT solving.

One recent promising area of research is the integration of extended resolution into SAT solvers [5, 53]. Extended resolution allows definitions to be created (e.g., new variables representing some Boolean expression). This can provide an added degree of flexibility in modern CDCL SAT solvers.

Another recent promising area of research is to use the DPLL(T) paradigm [88] in designing problem-specific SAT-based algorithms. Concrete examples include specific solvers for handling SAT problems with parity constraints [65], and also PBO solvers [71].

One additional area for future improvements to SAT solvers is formula simplification, before or during search [37, 57, 58].

Besides improvements to SAT solver technology, a number of additional research directions can be envisioned in the area of SAT solving. First, applications of SAT continue to be proposed on a regular basis. This is expected to continue in the future. A related topic is the development of improvements to existing applications of SAT. Moreover, the general area of SAT-based problem solving has been the subject of remarkable improvements in recent years, namely in terms of the many uses of SAT solvers as oracles for solving function problems. Concrete examples include Maximum Satisfiability (MaxSAT) and Pseudo-Boolean Optimization (PBO) [66, 95], minimal unsatisfiable subsets (MUSes) [16, 27], minimal correction subsets (MCSes) [75], backbones of Boolean formulas [77, 121], minimal models, and, in general, minimal sets over monotone predicates [76].

One final area of research is Quantified Boolean Formulas (QBF). Despite the many improvements made in recent years, improvements to QBF solvers are still far inferior to those made to SAT solvers. Nevertheless, recent new uses of SAT solvers in QBF solving suggest further improvements are to be expected [56].

References

1. Abío, I., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E., Mayer-Eichberger, V.: A new look at BDDs for pseudo-boolean constraints. J. Artif. Intell. Res. **45**, 443–480 (2012)
2. Achá, R.J.A., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E.: Practical algorithms for unsatisfiability proof and core generation in SAT solvers. AI Commun. **23**(2–3), 145–157 (2010)
3. Ansótegui, C., Bonet, M.L., Levy, J.: SAT-based MaxSAT algorithms. Artif. Intell. **196**, 77–105 (2013)
4. Asín, R., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E.: Cardinality networks: a theoretical and empirical study. Constraints **16**(2), 195–221 (2011)
5. Audemard, G., Katsirelos, G., Simon, L.: A restriction of extended resolution for clause learning SAT solvers. In: Fox, M., Poole, D. (eds.) AAAI Conf. on Artificial Intelligence (AAAI), pp. 15–20. AAAI Press, Palo Alto (2010)

6. Audemard, G., Simon, L.: Experimenting with small changes in conflict-driven clause learning algorithms. In: Stuckey, P.J. (ed.) Intl. Conf. on Principles and Practice of Constraint Programming (CP). LNCS, vol. 5202, pp. 630–634. Springer, Heidelberg (2008)
7. Audemard, G., Simon, L.: Predicting learnt clauses quality in modern SAT solvers. In: Boutilier, C. (ed.) Intl. Joint Conf. on Artificial Intelligence (IJCAI), pp. 399–404. IJCAI/AAAI Press, Melbourne/Palo Alto (2009)
8. Bailleux, O., Boufkhad, Y., Roussel, O.: New encodings of pseudo-boolean constraints into CNF. In: Kullmann, O. (ed.) Theory and Applications of Satisfiability Testing (SAT). LNCS, vol. 5584, pp. 181–194. Springer, Heidelberg (2009)
9. Baker, A.B.: The hazards of fancy backtracking. In: Hayes-Roth, B., Korf, R.E. (eds.) National Conference on Artificial Intelligence (AAAI), pp. 288–293. AAAI Press/MIT Press, Palo Alto/Cambridge (1994)
10. Bakker, R.R., Dikker, F., Tempelman, F., Wognum, P.M.: Diagnosing and solving overdetermined constraint satisfaction problems. In: Bajcsy, R. (ed.) Intl. Joint Conf. on Artificial Intelligence (IJCAI), pp. 276–281. Morgan Kaufmann, Cambridge (1993)
11. Baptista, L., Marques-Silva, J.: Using randomization and learning to solve hard real-world instances of satisfiability. In: Dechter [34], pp. 489–494
12. Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 825–885. IOS Press, Amsterdam (2009)
13. Batcher, K.E.: Sorting networks and their applications. In: AFIPS Spring Joint Computer Conference. AFIPS Conference Proceedings, vol. 32, pp. 307–314. Thomson Book Co., Washington D.C. (1968)
14. Bayardo, R.J. Jr., Schrag, R.: Using CSP look-back techniques to solve real-world SAT instances. In: Kuipers, B., Webber, B.L. (eds.) National Conference on Artificial Intelligence (AAAI), pp. 203–208. AAAI Press/MIT Press, Palo Alto/Cambridge (1997)
15. Belov, A., Chen, H., Mishchenko, A., Marques-Silva, J.: Core minimization in SAT-based abstraction. In: Macii, E. (ed.) Design, Automation & Test in Europe (DATE), pp. 1411–1416. EDA Consortium/ACM, San Jose/New York (2013)
16. Belov, A., Lynce, I., Marques-Silva, J.: Towards efficient MUS extraction. AI Commun. **25**(2), 97–116 (2012)
17. Ben-Eliyahu, R., Dechter, R.: On computing minimal models. Ann. Math. Artif. Intell. **18**(1), 3–27 (1996)
18. Ben-Eliyahu-Zohary, R., Palopoli, L.: Reasoning with minimal models: efficient algorithms and applications. Artif. Intell. **96**(2), 421–449 (1997)
19. Biere, A.: PicoSAT essentials. J. Satisf. Boolean Model. Comput. **4**(2–4), 75–97 (2008)
20. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, R. (ed.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
21. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press, Amsterdam (2009)
22. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D.A. (eds.) Intl. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI). LNCS, vol. 6538, pp. 70–87. Springer, Heidelberg (2011)
23. Bradley, A.R., Manna, Z.: Property-directed incremental invariant generation. Form. Asp. Comput. **20**(4–5), 379–405 (2008)
24. Brafman, R.I.: A simplifier for propositional formulas with many binary clauses. IEEE Trans. Syst. Man Cybern., Part B, Cybern. **34**(1), 52–59 (2004)
25. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. Trans. Comput. **35**(8), 677–691 (1986)
26. Büning, H.K., Bubeck, U.: Theory of quantified boolean formulas. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 735–760. IOS Press, Amsterdam (2009)

27. Büning, H.K., Kullmann, O.: Minimal unsatisfiability and autarkies. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 339–401. IOS Press, Amsterdam (2009)
28. Chinneck, J.W., Dravnieks, E.W.: Locating minimal infeasible constraint sets in linear programs. INFORMS J. Comput. **3**(2), 157–168 (1991)
29. Coelho, H., Studer, R., Wooldridge, M. (eds.): Proceedings of the ECAI 2010—19th European Conference on Artificial Intelligence, Lisbon, Portugal, August 16–20, 2010. IOS Press, Amsterdam (2010)
30. Cook, S.A.: The complexity of theorem-proving procedures. In: Harrison, M.A., Banerji, R.B., Ullman, J.D. (eds.) Annual Symp. on Theory of Computing (STOC), pp. 151–158. ACM, New York (1971)
31. Crawford, J.M., Auton, L.D.: Experimental results on the crossover point in satisfiability problems. In: Fikes, R., Lehnert, W.G. (eds.) National Conference on Artificial Intelligence (AAAI), pp. 21–27. AAAI Press/MIT Press, Palo Alto/Cambridge (1993)
32. Davis, M., Logemann, G., Loveland, D.W.: A machine program for theorem-proving. Commun. ACM **5**(7), 394–397 (1962)
33. Davis, M., Putnam, H.: A computing procedure for quantification theory. J. ACM **7**(3), 201–215 (1960)
34. Dechter, R. (ed.): Principles and Practice of Constraint Programming—CP 2000, Proceedings of the 6th International Conference, Singapore, September 18–21, 2000. LNCS, vol. 1894. Springer, Heidelberg (2000)
35. Dershowitz, N., Hanna, Z., Nadel, A.: A scalable algorithm for minimal unsatisfiable core extraction. In: Biere, A., Gomes, C.P. (eds.) Theory and Applications of Satisfiability Testing (SAT). LNCS, vol. 4121, pp. 36–41. Springer, Heidelberg (2006)
36. Dubois, O., André, P., Boufkhad, Y., Carlier, J.: SAT versus UNSAT. In: Johnson, D.S., Trick, M. (eds.) Cliques, Coloring and Satisfiability: Second DIMACS Implementation Challenge. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 26, pp. 415–436. AMS, Providence (1996)
37. Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In: Bacchus, F., Walsh, T. (eds.) Theory and Applications of Satisfiability Testing (SAT). LNCS, vol. 3569, pp. 61–75. Springer, Heidelberg (2005)
38. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) Theory and Applications of Satisfiability Testing (SAT). LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2003)
39. Eén, N., Sörensson, N.: Translating pseudo-boolean constraints into SAT. J. Satisf. Boolean Model. Comput. **2**(1–4), 1–26 (2006)
40. Freeman, J.W.: Improvements to propositional satisfiability search algorithms. Ph.D. thesis, University of Pennsylvania (1995)
41. Frisch, A.M., Peugniez, T.J.: Solving non-boolean satisfiability problems with stochastic local search. In: Nebel, B. (ed.) Intl. Joint Conf. on Artificial Intelligence (IJCAI), pp. 282–290. Morgan Kaufmann, Cambridge (2001)
42. Fujiwara, H., Shimono, T.: On the acceleration of test generation algorithms. Trans. Comput. **32**(12), 1137–1144 (1983)
43. Giunchiglia, E., Marin, P., Narizzano, M.: Reasoning with quantified boolean formulas. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 761–780. IOS Press, Amsterdam (2009)
44. Goldberg, E.I., Novikov, Y.: BerkMin: a fast and robust Sat-solver. In: Design, Automation & Test in Europe (DATE), pp. 142–149. IEEE, Piscataway (2002)
45. Gomes, C.P., Selman, B., Kautz, H.A.: Boosting combinatorial search through randomization. In: Mostow, J., Rich, C. (eds.) National Conference on Artificial Intelligence (AAAI), pp. 431–437. AAAI Press/MIT Press, Palo Alto/Cambridge (1998)

46. Grégoire, É., Mazure, B., Piette, C.: On approaches to explaining infeasibility of sets of boolean clauses. In: Intl. Conf. on Tools with Artificial Intelligence (ICTAI), vol. 1, pp. 74–83. IEEE, Piscataway (2008)
47. Guo, L., Hamadi, Y., Jabbour, S., Sais, L.: Diversification and intensification in parallel SAT solving. In: Cohen, D. (ed.) Intl. Conf. on Principles and Practice of Constraint Programming (CP). LNCS, vol. 6308, pp. 252–265. Springer, Heidelberg (2010)
48. Hamadi, Y., Jabbour, S., Sais, L.: ManySAT: a parallel SAT solver. J. Satisf. Boolean Model. Comput. **6**(4), 245–262 (2009)
49. Hamadi, Y., Wintersteiger, C.M.: Seven challenges in parallel SAT solving. AI Mag. **34**(2), 99–106 (2013)
50. Hemery, F., Lecoutre, C., Sais, L., Boussemart, F.: Extracting MUCs from constraint networks. In: Brewka, G., Coradeschi, S., Perini, A., Traverso, P. (eds.) European Conf. on Artificial Intelligence (ECAI), pp. 113–117. IOS Press, Amsterdam (2006)
51. Heras, F., Larrosa, J., Oliveras, A.: MiniMaxSAT: an efficient weighted Max-SAT solver. J. Artif. Intell. Res. **31**, 1–32 (2008)
52. Huang, J.: The effect of restarts on the efficiency of clause learning. In: Veloso, M.M. (ed.) Intl. Joint Conf. on Artificial Intelligence (IJCAI), pp. 2318–2323 (2007)
53. Huang, J.: Extended clause learning. Artif. Intell. **174**(15), 1277–1284 (2010)
54. Jackson, D., Vaziri, M.: Finding bugs with a constraint solver. In: Intl. Symp. on Software Testing and Analysis (ISSTA), pp. 14–25. ACM, New York (2000)
55. Janhunen, T., Niemelä, I.: Compact translations of non-disjunctive answer set programs to propositional clauses. In: Balduccini, M., Son, T.C. (eds.) Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning—Essays Dedicated to Michael Gelfond on the Occasion of His 65th Birthday. Lecture Notes in Artificial Intelligence, vol. 6565, pp. 111–130. Springer, Heidelberg (2011)
56. Janota, M., Klieber, W., Marques-Silva, J., Clarke, E.M.: Solving QBF with counterexample guided refinement. In: Cimatti, A., Sebastiani, R. (eds.) Theory and Applications of Satisfiability Testing (SAT). LNCS, vol. 7317, pp. 114–128. Springer, Heidelberg (2012)
57. Järvisalo, M., Biere, A., Heule, M.: Simulating circuit-level simplifications on CNF. J. Autom. Reason. **49**(4), 583–619 (2012)
58. Järvisalo, M., Heule, M., Biere, A.: Inprocessing rules. In: Gramlich, B., Miller, D., Sattler, U. (eds.) Intl. Joint Conf. on Automated Reasoning (IJCAR). LNCS, vol. 7364, pp. 355–370. Springer, Heidelberg (2012)
59. Jin, H., Han, H., Somenzi, F.: Efficient conflict analysis for finding all satisfying assignments of a boolean circuit. In: Halbwachs, N., Zuck, L.D. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 3440, pp. 287–300. Springer, Heidelberg (2005)
60. Junker, U.: QuickXplain: preferred explanations and relaxations for over-constrained problems. In: McGuinness, D.L., Ferguson, G. (eds.) National Conference on Artificial Intelligence (AAAI), pp. 167–172. AAAI Press/MIT Press, Palo Alto/Cambridge (2004)
61. Kautz, H.A., Selman, B.: Planning as satisfiability. In: Neumann, B. (ed.) European Conf. on Artificial Intelligence (ECAI), pp. 359–363. Wiley, New York (1992)
62. Kleine-Büning, H., Letterman, T.: Propositional Logic: Deduction and Algorithms. Cambridge University Press, Cambridge (1999)
63. Kroening, D., Strichman, O.: Decision Procedures: An Algorithmic Point of View. Springer, Heidelberg (2008)
64. Kullmann, O. (ed.): Theory and Applications of Satisfiability Testing—SAT 2009, Proceedings of the 12th International Conference, SAT 2009, Swansea, UK, June 30–July 3, 2009. LNCS, vol. 5584. Springer, Heidelberg (2009)
65. Laitinen, T., Junttila, T.A., Niemelä, I.: Extending clause learning DPLL with parity reasoning. In: Coelho et al. [29], pp. 21–26
66. Li, C.M., Manyà, F.: MaxSAT, hard and soft constraints. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 613–631. IOS Press, Amsterdam (2009)

67. Liffiton, M.H., Sakallah, K.A.: Algorithms for computing minimal unsatisfiable subsets of constraints. J. Autom. Reason. **40**(1), 1–33 (2008)
68. Lynce, I., Baptista, L., Marques-Silva, J.: Towards provably complete stochastic search algorithms for satisfiability. In: Brazdil, P., Jorge, A. (eds.) Portuguese Conf. on Artificial Intelligence (EPIA). LNCS, vol. 2258, pp. 363–370. Springer, Heidelberg (2001)
69. Lynce, I., Marques-Silva, J.: Probing-based preprocessing techniques for propositional satisfiability. In: Intl. Conf. on Tools with Artificial Intelligence (ICTAI), pp. 105–110. IEEE, Piscataway (2003)
70. Lynce, I., Marques-Silva, J.: Efficient data structures for backtrack search SAT solvers. Ann. Math. Artif. Intell. **43**(1), 137–152 (2005)
71. Manquinho, V.M., Marques-Silva, J.: Satisfiability-based algorithms for boolean optimization. Ann. Math. Artif. Intell. **40**(3–4), 353–372 (2004)
72. Marques-Silva, J.: Search algorithms for satisfiability problems in combinational switching circuits. Ph.D. thesis, University of Michigan (1995)
73. Marques-Silva, J.: The impact of branching heuristics in propositional satisfiability algorithms. In: Barahona, P., Alferes, J.J. (eds.) Portuguese Conf. on Artificial Intelligence (EPIA). LNCS, vol. 1695, pp. 62–74. Springer, Heidelberg (1999)
74. Marques-Silva, J.: Algebraic simplification techniques for propositional satisfiability. In: Dechter [34], pp. 537–542
75. Marques-Silva, J., Heras, F., Janota, M., Previti, A., Belov, A.: On computing minimal correction subsets. In: Rossi, F. (ed.) Intl. Joint Conf. on Artificial Intelligence (IJCAI), pp. 615–622. IJCAI/AAAI, Melbourne/Palo Alto (2013)
76. Marques-Silva, J., Janota, M., Belov, A.: Minimal sets over monotone predicates in boolean formulae. In: Sharygina, N., Veith, H. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 8044, pp. 592–607. Springer, Heidelberg (2013)
77. Marques-Silva, J., Janota, M., Lynce, I.: On computing backbones of propositional theories. In: Coelho et al. [29], pp. 15–20
78. Marques-Silva, J., Lynce, I., Malik, S.: Conflict-driven clause learning SAT solvers. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 131–153. IOS Press, Amsterdam (2009)
79. Marques-Silva, J., Sakallah, K.A.: Dynamic search-space pruning techniques in path sensitization. In: Lorenzetti, M.J. (ed.) Design Automation Conf. (DAC), pp. 705–711. ACM, New York (1994)
80. Marques-Silva, J., Sakallah, K.A.: GRASP—a new search algorithm for satisfiability. In: Rutenbar, R.A., Otten, R.H.J.M. (eds.) Intl. Conf. on Computer-Aided Design (ICCAD), pp. 220–227. IEEE/ACM, Piscataway/New York (1996)
81. Marques-Silva, J., Sakallah, K.A.: GRASP: a search algorithm for propositional satisfiability. Trans. Comput. **48**(5), 506–521 (1999)
82. Marques-Silva, J., Sakallah, K.A. (eds.): Theory and Applications of Satisfiability Testing—SAT 2007, Proceedings of the 10th International Conference, Lisbon, Portugal, May 28–31, 2007. LNCS, vol. 4501. Springer, Heidelberg (2007)
83. McMillan, K.L.: Applying SAT methods in unbounded symbolic model checking. In: Brinksma, E., Larsen, K.G. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 2404, pp. 250–264. Springer, Heidelberg (2002)
84. McMillan, K.L.: Interpolation and SAT-based model checking. In: Hunt, W.A. Jr., Somenzi, F. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 2725, pp. 1–13. Springer, Heidelberg (2003)
85. Morgado, A., Heras, F., Liffiton, M.H., Planes, J., Marques-Silva, J.: Iterative and core-guided MaxSAT solving: a survey and assessment. Constraints **18**(4), 478–534 (2013)
86. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient SAT solver. In: Design Automation Conf. (DAC), pp. 530–535. ACM, New York (2001)
87. Nam, G.J., Sakallah, K.A., Rutenbar, R.A.: Satisfiability-based detailed FPGA routing. In: Intl. Conf. on VLSI Design, pp. 574–577. IEEE, Piscataway (1999)

88. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories: from an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). J. ACM **53**(6), 937–977 (2006)
89. Pipatsrisawat, K., Darwiche, A.: A lightweight component caching scheme for satisfiability solvers. In: Marques-Silva and Sakallah [82], pp. 294–299
90. Plaisted, D.A., Greenbaum, S.: A structure-preserving clause form translation. J. Symb. Comput. **2**(3), 293–304 (1986)
91. Prestwich, S.D.: Variable dependency in local search: prevention is better than cure. In: Marques-Silva and Sakallah [82], pp. 107–120
92. Prestwich, S.D.: CNF encodings. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 75–97. IOS Press, Amsterdam (2009)
93. Ravi, K., Somenzi, F.: Minimal assignments for bounded model checking. In: Jensen, K., Podelski, A. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 2988, pp. 31–45. Springer, Heidelberg (2004)
94. Robinson, J.A.: A machine-oriented logic based on the resolution principle. J. ACM **12**(1), 23–41 (1965)
95. Roussel, O., Manquinho, V.M.: Pseudo-boolean and cardinality constraints. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 695–733. IOS Press, Amsterdam (2009)
96. Sabharwal, A., Samulowitz, H., Sellmann, M.: Learning back-clauses in SAT. In: Cimatti, A., Sebastiani, R. (eds.) Theory and Applications of Satisfiability Testing (SAT). LNCS, vol. 7317, pp. 498–499. Springer, Heidelberg (2012)
97. Schulz, M.H., Trischler, E., Sarfert, T.M.: SOCRATES: a highly efficient automatic test pattern generation system. Trans. Comput.-Aided Des. Integr. Circuits Syst. **7**(1), 126–137 (1988)
98. Selman, B., Kautz, H., Cohen, B.: Local search strategies for satisfiability testing. In: Johnson, D.S., Trick, M. (eds.) Cliques, Coloring and Satisfiability: Second DIMACS Implementation Challenge. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 26, pp. 521–532. AMS, Providence (1996)
99. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In: Hunt, W.A. Jr., Johnson, S.D. (eds.) Formal Methods in Computer Aided Design (FMCAD). LNCS, vol. 1954, pp. 108–125. Springer, Heidelberg (2000)
100. Sheeran, M., Stålmarck, G.: A tutorial on Stålmarck's proof procedure for propositional logic. Form. Methods Syst. Des. **16**(1), 23–58 (2000)
101. Shlyakhter, I., Seater, R., Jackson, D., Sridharan, M., Taghdiri, M.: Debugging overconstrained declarative models using unsatisfiable cores. In: Intl. Conf. on Automated Software Engineering (ASE), pp. 94–105. IEEE, Piscataway (2003)
102. Sinz, C.: Towards an optimal CNF encoding of boolean cardinality constraints. In: van Beek, P. (ed.) Intl. Conf. on Principles and Practice of Constraint Programming (CP). LNCS, vol. 3709, pp. 827–831. Springer, Heidelberg (2005)
103. Sinz, C., Iser, M.: Problem-sensitive restart heuristics for the DPLL procedure. In: Kullmann [64], pp. 356–362
104. de Siqueira, J.L.N., Puget, J.F.: Explanation-based generalisation of failures. In: Kodratoff, Y. (ed.) European Conf. on Artificial Intelligence (ECAI), pp. 339–344. Pitman, London (1988)
105. Sörensson, N., Biere, A.: Minimizing learned clauses. In: Kullmann [64], pp. 237–243
106. Stallman, R.M., Sussman, G.J.: Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. Artif. Intell. **9**(2), 135–196 (1977)
107. Stephan, P.R., Brayton, R.K., Sangiovanni-Vincentelli, A.L.: Combinational test generation using satisfiability. Trans. Comput.-Aided Des. Integr. Circuits Syst. **15**(9), 1167–1176 (1996)

108. Stuckey, P.J.: There are no CNF problems. In: Järvisalo, M., Gelder, A.V. (eds.) Theory and Applications of Satisfiability Testing (SAT). LNCS, vol. 7962, pp. 19–21. Springer, Heidelberg (2013)
109. Tarjan, R.E.: Finding dominators in directed graphs. SIAM J. Comput. **3**(1), 62–89 (1974)
110. Tseitin, G.S.: On the complexity of derivation in propositional calculus. In: Silenko, A.O. (ed.) Studies in Constructive Mathematics and Mathematical Logic, Part II, pp. 115–125. Springer, Heidelberg (1968)
111. Urquhart, A.: The complexity of propositional proofs. Bull. Symb. Log. **1**(4), 425–467 (1995)
112. Van Gelder, A., Tsuji, Y.K.: Satisfiability testing with more reasoning and less guessing. Tech. rep., University of California at Santa Cruz (1995)
113. Warners, J.P.: A linear-time transformation of linear inequalities into conjunctive normal form. Inf. Process. Lett. **68**(2), 63–69 (1998)
114. Wieringa, S.: Understanding, improving and parallelizing MUS finding using model rotation. In: Milano, M. (ed.) Intl. Conf. on Principles and Practice of Constraint Programming (CP). LNCS, vol. 7514, pp. 672–687. Springer, Heidelberg (2012)
115. Wolfram, D.A.: Forward checking and intelligent backtracking. Inf. Process. Lett. **32**(2), 85–87 (1989)
116. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: SATzilla: Portfolio-based algorithm selection for SAT. J. Artif. Intell. Res. **32**, 565–606 (2008)
117. Zabih, R., McAllester, D.A.: A rearrangement search strategy for determining propositional satisfiability. In: Shrobe, H.E., Mitchell, T.M., Smith, R.G. (eds.) National Conference on Artificial Intelligence (AAAI), pp. 155–160. AAAI Press/MIT Press, Palo Alto/Cambridge (1988)
118. Zhang, H.: SATO: an efficient propositional prover. In: McCune, W. (ed.) Intl. Conf. on Automated Deduction (CADE). LNCS, vol. 1249, pp. 272–275. Springer, Heidelberg (1997)
119. Zhang, H., Stickel, M.E.: Implementing the Davis-Putnam method. J. Autom. Reason. **24**(1/2), 277–296 (2000)
120. Zhang, L., Malik, S.: Validating SAT solvers using an independent resolution-based checker: practical implementations and other applications. In: Design, Automation & Test in Europe (DATE), pp. 10,880–10,885. IEEE, Piscataway (2003)
121. Zhu, C.S., Weissenbacher, G., Sethi, D., Malik, S.: SAT-based techniques for determining backbones for post-silicon fault localisation. In: Zilic, Z., Shukla, S.K. (eds.) Intl. High Level Design Validation and Test Workshop (HLDVT), pp. 84–91. IEEE, Piscataway (2011)

Chapter 10
SAT-Based Model Checking

Armin Biere and Daniel Kröning

Abstract Modern satisfiability (SAT) solvers have become the enabling technology of many model checkers. In this chapter, we will focus on those techniques most relevant to industrial practice. In *bounded model checking* (BMC), a transition system and a property are jointly unwound for a given number k of steps to obtain a formula that is satisfiable if there is a counterexample for the property up to length k. The formula is then passed to an efficient SAT solver. The strength of BMC is *refutation*: BMC has been used to discover subtle flaws in digital systems. We cover the application of BMC to both hardware and software systems, and to hardware/software co-verification. We also discuss means to make BMC complete, including k-induction, Craig interpolation, abstraction refinement techniques, and inductive techniques with iterative strengthening.

10.1 Introduction

Modern satisfiability (SAT) solvers have become the core technology of many model checkers, greatly improving capacity when compared to BDD-based model checkers. In this chapter, we will focus on those SAT-based model-checking techniques that are most relevant to industrial practice. In SAT-based *bounded model checking* (BMC) [26], a symbolic representation of a transition system and a property are jointly unwound for a given number of steps k to obtain a formula that is satisfiable if there is a counterexample for the property up to length k. The formula is then passed to an efficient SAT solver.

The idea of using propositional SAT to encode and solve path constraints for transition systems was discussed before in the AI planning community. Originally Kautz and Selman [103] observed that direct encodings of planning problems into a propositional SAT problem outperformed the best planning algorithms by orders

A. Biere (✉)
Johannes Kepler University, Linz, Austria
e-mail: biere@jku.at

D. Kröning
University of Oxford, Oxford, UK

E.M. Clarke et al. (eds.), *Handbook of Model Checking*,
DOI 10.1007/978-3-319-10575-8_10

of magnitude. A more recent experimental survey of using SAT for planning can be found in [145].

The rationale for using BMC is based on the observation that SAT solvers are often able to solve much larger formulas than classical techniques based on binary decision diagrams (BDDs) [40] (see also Chap. 8 of this Handbook). It is now industrial practice to simply run BMC for a certain amount of time or up to a certain bound k, fixed for instance in the verification plan.

On the other hand, BDD-based techniques allow efficient implementations of quantifier elimination, which is crucial for termination checks in symbolic fix-point algorithms. The detection of the fix-point is essential to *prove* properties in general, but not necessary when aiming at *refutation*.

In this chapter, we cover the application of BMC to both hardware and software systems, and hardware/software co-verification. In its simplest form, BMC is incomplete, as bugs that are only exposed with more than k transitions are missed. These BMC-based techniques therefore either relinquish completeness, or have to rely on alternative ways to assert that a property holds in general for all bounds. The chapter therefore covers a range of SAT-based techniques that are able to establish a proof of correctness for the property for an unbounded depth.

This material has been covered more extensively in other tutorial-style publications and surveys before [69, 81, 141, 155], including two chapters [24, 110] in the *Handbook of Satisfiability* [28], by the same authors as this chapter. Thus, besides explaining some of the very basic ideas, the rather restricted amount of space available here is used to give pointers to existing important work on SAT-based model checking and elaborating on more recent publications.

The outline of the chapter is as follows. We begin with a description of how to perform BMC on an abstract description of the system, given in the form of a *transition system*. We then provide details on how to obtain formal models from industrial system description languages such as Verilog and ANSI-C, and how to encode these models and systems properties into a propositional formula. In particular, we show how model-checking problems for software and hardware can be encoded into satisfiability checking (SAT). The chapter concludes with a discussion of means to make BMC complete, including k-induction, Craig interpolation, and inductive techniques with iterative strengthening.

10.2 Bounded Model Checking on Kripke Structures

10.2.1 Kripke Structures

The behaviors of a program or circuit can be formally captured using a *Kripke Structure*, formally defined as follows.

Definition 1 (Kripke structure) A *Kripke Structure* is a (finite) set of states S, a set of initial states $I \subseteq S$, and a transition relation $T \subseteq S \times S$.

A *path* in a Kripke structure is a (possibly infinite) sequence of states $s_0, s_1, s_2, \ldots$ such that

- s_0 is an initial state, i.e., $s_0 \in I$, and
- there is a transition between any s_i and s_{i+1}, i.e., $(s_i, s_{i+1}) \in T$.

The states are typically valuations of a set of *state variables*, corresponding to latches and registers in circuits and program variables in software. In the case of a finite set of states we can always re-encode the Kripke structure to use propositional variables only. As a result, we obtain purely propositional predicates I and T. We use the set notation and the state predicates and relations interchangeably, i.e., the propositional formula $I(s_i)$ evaluates to true iff $s_i \in I$. Similarly, $T(s_i, s_{i+1})$ evaluates to true iff $(s_i, s_{i+1}) \in T$.

The key idea of bounded model checking is to construct a formula that is satisfiable if there exists a path that violates a given property. We now consider specific kinds of properties, and will distinguish *safety* and *liveness* properties.

10.2.2 Safety Properties

Properties are typically defined using a suitable *temporal logic*. We refer to Chap. 2 of this Handbook [140] for an introduction to temporal logics. We restrict the discussion in this chapter to properties given in *Linear Temporal Logic* (LTL). One benefit of this restriction is that counterexamples to LTL properties can always be given in the form of a path, as defined above. A full survey on ways to encode LTL in a BMC context together with an experimental comparison with BDD-based techniques is provided by Biere et al. [27]. Note that encodings differ in terms of compactness, ease of implementation, and of course SAT-solving efficiency.

We begin with LTL properties of the form $\mathbf{G}\,p$, where p is a state predicate. This property establishes that p is a global invariant of the system. A counterexample for a property of this kind can be given as a finite path that ends with a state s that satisfies $\neg p$. This gives rise to a straightforward condition for the existence of a counterexample path of length k:

$$\exists s_0, \ldots, s_k. \quad I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \neg p(s_k) \tag{1}$$

The formula above contains three conjuncts. The first conjunct, $I(s_0)$, ensures that the state s_0 is one of the initial states. The second conjunct encodes the requirement that there is a transition from s_i to s_{i+1} for each $i \in \{0, \ldots, k-1\}$. This amounts to creating k replicas of the transition relation T. Finally, the conjunct $\neg p(s_k)$ asserts that the state s_k satisfies $\neg p$.

Note that the formula obtained in this way has only one level of (existential) quantification and thus corresponds to a propositional satisfiability problem. Most modern SAT solvers such as ZChaff [136] or MiniSAT [73] expect to receive the

propositional formula in conjunctive normal form (CNF).[1] The transformation of the quantifier-free propositional formula into CNF is performed using the *Tseitin transformation* [151]. This transformation is linear-time, and results in an equisatisfiable formula in CNF. Numerous papers on more compact or more efficient variants of this step have been published, e.g., [45, 72, 153]. Further details on CNF encodings can also be found in the *Handbook of Satisfiability* [142]. See also the discussion on the relation between CNF-level preprocessing and encoding in [97].

10.2.3 Liveness Properties

We will consider further categories of system properties. The simplest type of liveness properties are *eventualities*, e.g., whether a particular state property is guaranteed to eventually hold. These properties are written as $\mathbf{F}p$ in LTL. The encoding of LTL formulas of this form is very similar to the encoding of $\mathbf{G}p$. We observe that counterexamples to properties of this form can always be given as a finite (possibly empty) prefix (called the *stem*) followed by a finite loop. All states on the path satisfy $\neg p$. This pattern can be encoded as follows:

$$\exists s_0, \ldots, s_k. \quad I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \bigwedge_{i=0}^{k-1} \neg p(s_i) \wedge \bigvee_{i=0}^{k-1} s_k = s_i \tag{2}$$

As described above, the formula can be converted into propositional logic, and can then be passed to a propositional SAT solver.

The translation of general LTL formulas is more complex. Techniques for performing this translation can be categorized as *syntactic* or *semantic* [58]. Syntactic translations follow the syntactic structure of the LTL property; instances include [26, 90, 91, 128, 139].

As an alternative *semantic translations* can be used, which are based on automata: the formula is transformed into a suitable kind of automaton that accepts counterexample paths. An instance is the translation of the LTL property φ into a Büchi automaton $M_{\neg\varphi}$ that accepts paths that satisfy $\neg\varphi$ [74, 152]. Counterexamples to φ then have the form of a path through the product of the Kripke structure and $M_{\neg\varphi}$ that contains infinitely many accepting states. A counterexample in a finite-state product is thus a loop that does not contain an accepting state. This condition can be encoded using a formula similar to Eq. (2). One key advantage of the automata-based encoding is that numerous minimization techniques can be applied to the automaton prior to building the BMC formula.

Semantic translations allow the use of sophisticated automata optimization techniques, but the space requirements might explode for larger formulas, due to explicit representation of potentially exponentially many states in the automata.

[1] There are now also non-clausal propositional SAT solvers, e.g., [95].

10.2.3.1 Liveness to Safety Translation

Besides these syntactic and semantic translations, a third approach to handle liveness is to encode liveness into safety (L2S) and then use model-checking algorithms for checking safety [25, 146]. This is particularly useful for techniques such as interpolation which only work for safety properties at this point. The L2S encoding actually increases the size of the model by a factor of two. Thus, it might be prohibitively expensive for BDD-based techniques, which are very sensitive to model size. However, even for BDD-based model checking there are cases where L2S is exponentially more efficient.

10.2.3.2 *k*-Liveness

More recently, a new approach for checking liveness was presented in [49] and independently discovered in [80] (see also [124]). In [49], the authors called it "*k*-liveness". Their implementation proved to be quite effective in the liveness track of the Hardware Model Checking Competition 2012 (HWMCC 2012). In this approach, liveness properties are assumed to be encoded as $\mathbf{FG}p$ properties. Then the approach tries to prove that a witness trace for such a property does not exist. In case of a finite-state system, a witness trace to $\mathbf{FG}p$ can be assumed to be an infinite path which ends in a loop, where the loop contains a state in which p holds. If $\mathbf{FG}p$ cannot be satisfied, then the prefix of any path satisfies p only an arbitrary (but finite) number of times.

The basic idea of the approach is to count the number of occurrences of p and then check that the count is smaller than a fixed bound k. Note that this turns the liveness-checking problem into a simple safety-checking problem. If p can only be satisfied at most k times, then $\mathbf{FG}p$ cannot be satisfied on any initialized path. If the safety check fails and a path is found on which p can be satisfied more than k times, the bound k is increased to say $k+1$ and a new safety-checking problem for bound $k+1$ is generated. If the property $\mathbf{FG}p$ does not hold for a finite-state system, then this process has to terminate after k reaches the number of states of the system. In practice the process terminates much earlier, in particular if combined with a method for extracting additional constraints [49]. In order to find violations of liveness properties, i.e., witness traces for formulas like $\mathbf{FG}p$, the approach has to rely on other techniques, such as those discussed above.

10.3 Bounded Model Checking for Hardware Designs

We will now cover techniques to translate system descriptions given in industrial system description languages into BMC instances. We begin with verification of designs given in hardware description languages (HDLs), which was one of the earliest applications of SAT-based BMC (see also Chap. 24 of this Handbook [77]).

10.3.1 Hardware Description Languages (HDLs)

In industrial practice, hardware designs are described by means of modeling languages. These include languages to describe schematics and net-lists at the lowest level. Higher levels of abstraction can be achieved by hardware description languages (HDLs) such as *VHDL* or *Verilog*.

The challenges in encoding models given in hardware description languages into SAT are mostly shared by all model-checking techniques for hardware; they affect BDD-based and SAT-based methods alike. Most HDLs have both *simulation semantics* and *synthesis semantics*. Designers rely heavily on simulation and build models with simulation semantics in mind. Simulation semantics are typically based on an *event queue*, resembling the data structures maintained by event-driven simulators. On the other hand, the synthesis semantics is closer to the actual hardware produced, and may uncover design flaws that go unnoticed during simulation.

10.3.2 BMC on Net-Lists

We will briefly elaborate on performing BMC using synthesis semantics. In this context, the BMC implementation will initially perform several stages of behavioral synthesis up to the point that a *net-list* is produced. A net-list is a collection of primitive elements. A typical way to represent net-lists is to use an *and-inverter graph* (AIG) [123], i.e., the net-list consists of "and" gates, inverters and memory elements referred to as registers.

Definition 2 A net-list N is a directed graph (V_N, E_N, τ_N) where V_N is a finite set of vertices, $E_N \subseteq V_N \times V_N$ is the set of directed edges and $\tau_N : V_N \to \{\text{AND}, \text{INV}, \text{REG}, \text{INPUT}\}$ maps a node to its type, where AND is an "and" gate, INV is an inverter, REG is a register, and INPUT is a primary input. The in-degree of a vertex of type AND is at least two, of type INV and REG is exactly one and of type INPUT is zero. Any cycle in N must contain at least one REG node.

As an example, consider the 3-bit counter whose Verilog module is shown in Fig. 1 (taken from [47]). The corresponding net-list is shown in Fig. 2. A node drawn as a box represents a REG. A circle-shaped node is an AND gate. An incoming edge of a node marked with a circle indicates negation.

A *state* of a net-list is a mapping of its registers to the Boolean values $\mathbb{B} = \{0, 1\}$. A net-list N with r registers gives rise to a Kripke structure $M = (S_N, I_N, T_N)$ where $S_N = \mathbb{B}^r$ is the set of states and T_N is the transition relation specifying what pairs of states are connected by transitions. The set I_N of *initial states* is determined by the values of the registers immediately after reset. In the above example, $I_N = \neg count[0] \wedge \neg count[1] \wedge \neg count[2]$. The state-transition diagram for the circuit is shown in Fig. 3. Note that S_N for the 3-bit counter consists of $2^3 = 8$ states.

Fig. 1 Verilog module of a counter

```
module counter(clk, count);
  input clk;
  output [2:0] count;
  reg [2:0] count;

  wire cin =
    ~count[0] & ~count[1] & ~count[2];

  initial count = 3'b0;

  always @ (posedge clk) begin
       count[0] <= cin;
       count[1] <= count[0];
       count[2] <= count[1];
  end
endmodule
```

Fig. 2 Net-list for Fig. 1

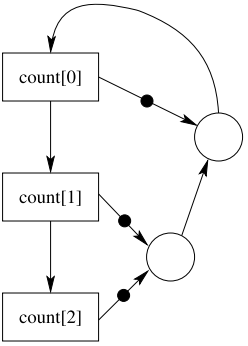
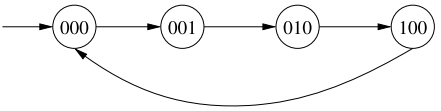


Fig. 3 State-transition diagram of the counter in Fig. 1



Unreachable states are not shown in Fig. 3. An algorithm for obtaining a transition relation for a net-list is given in [57].

The required property of a circuit can be given as part of the design description in languages such as *PSL* [76] or as a *System Verilog Assertion* [154]. A discussion of hardware specification languages can also be found in Chap. 24.

10.4 Bounded Model Checking for Software

We focus on BMC-like approaches to software verification; for a broader perspective on automated techniques for formal software verification, we refer the reader to a survey [69].

10.4.1 Monolithic Encodings

The most straightforward manner to implement BMC for software is to encode the transition relation of the program into a circuit representation, and then to perform BMC as described in Sect. 10.2.

1. The first step is to add a *program counter* (PC) to the set of state variables of the model. The program counter determines the instruction that is to be executed next.
2. Each instruction is turned separately into a transition relation. One way to obtain such a formula is to convert the arithmetic operators in the program into their circuit (net-list) equivalents. Arrays and pointers are treated as memories, using a large case split over the possible values of the address or a first-order array theory.

We will illustrate the second step by means of an example. Suppose that our program has three state variables named `x`, `y`, and `z`, and suppose we wish to encode the following instruction, given in C syntax:

$$\mathtt{x} = \mathtt{y} + 1;$$

Note that the equal sign = in the C program fragment above indicates an assignment, and not an equality relation. Following the usual convention, we will use $\mathtt{x}'$, $\mathtt{y}'$ and $\mathtt{z}'$ to denote the next-state values of the state variables. The transition relation for the statement above is then

$$\mathtt{x}' = \mathtt{y}' + 1 \quad \wedge \quad \mathtt{y}' = \mathtt{y} \quad \wedge \quad \mathtt{z}' = \mathtt{z}\,.$$

Note that in the above formula, the symbol $=$ denotes mathematical equality, and not assignment. Also note the second and third conjuncts: these constraints state the fact that the value of the program variables `y` and `z` is not changed by the instruction.

An unwinding using the "monolithic encoding" as described above with k steps permits all program paths that traverse k (or fewer) instructions to be explored. The size of this basic unwinding is k times the size of the program. For large programs, this is prohibitive, and thus, several optimizations have been proposed. These optimizations focus on reducing the size of the encoding by eliminating combinations of control-flow locations that do not correspond to paths through the program.

As an instance, in the case of sequential programs it is beneficial to merge all instructions within one basic block into a single big-step instruction. Each basic block of the program is converted into a formula by transforming it into *static single assignment* (SSA) form [3]. This reduces the number of control-flow locations. The model checker F-SOFT is reported to use an optimized monolithic encoding [94].

10.4.2 Path-Based Encodings

Instead of unwinding the entire transition relation, path-based software analyzers perform forward *symbolic execution* [105] or in general *symbolic simulation* alongside specific program paths up to a given depth. The resulting formula is then passed

to the SAT solver [62]. This basic approach has a broad range of applications; e.g., it can be used to check arbitrary safety properties or to generate test vectors to achieve particular coverage goals. See [43] for a historical perspective on symbolic execution.

There are numerous approaches to prune the set of paths that are to be explored, or to heuristically choose a path that most likely leads to a particular goal [14]. Once a satisfying assignment is obtained, a counterexample can be extracted. There is also work on obtaining particularly desirable counterexamples, and attempts to use information from the BMC instance to explain the root cause of the error [85, 87].

In the most basic form, tools using path-based encodings explore precisely one path at a time. An advantage of this approach is that the formulas generated this way are often very simple, and can be solved effectively by modern solvers. However, this basic approach to path-based exploration suffers from the *path explosion problem*, as the number of paths through a program is exponential in the worst case. As an example, consider a loop with a branch in the body. The branching decision is potentially independent in each iteration of the loop, and thus, the program has 2^n distinct paths for n loop iterations.

A principal method to address the path explosion problem is *path merging*. The idea is to merge the formulas that correspond to two (or more) paths at points of reconverging control flow. As a result, the number of formulas is reduced, but the resulting formulas are larger and thus more difficult to solve for the SAT solver. This enables a trade-off between the number of formulas to solve and their relative difficulty.

At the extreme end, the CBMC bounded model checker *always* merges, and thus, generates only a *single* formula for a given unwinding bound k [51, 52, 108, 112]. This formula is linear in the size of the program and linear in k even if there is an exponential number of paths in the program. This corresponds to replicating the basic blocks along the path k times, followed by a transformation of the concatenation of these blocks into SSA form [3]. Other tools perform path merging heuristically in order to contain the total number of formulas.

10.4.3 Completeness for Bounded Programs

Bounded model checking, when applied as described above, is inherently incomplete, as it searches for property violations only up to a given bound and never returns "*No Errors*". Bugs that are deeper than the given bound are missed. Nevertheless, BMC can be used to *prove* liveness and safety properties on a particular class of programs if applied in a slightly different way. The class we consider here are programs that have a high-level worst-case execution time (WCET). Numerous programs are required to have this property, especially in the domain of safety-critical embedded software.

A high-level WCET is typically given by a bound on the maximum number of loop iterations and is usually computed via a simple syntactic analysis of loop structures. If the syntactic analysis fails, an iterative algorithm can be applied. First,

a guess k for the bound on the number of loop iterations is made. The loop is then unrolled up to this bound k using BMC. The property that is checked is that any path exceeding k loop iterations is infeasible. If the property holds, k is established as a sound high-level WCET. Otherwise, there are paths in the program exceeding the bound, and a new guess for the bound is made [52, 112].

10.4.4 BMC for Multi-threaded Programs

The verification of concurrent software is primarily discussed in Chap. 18 of this Handbook [88]. We will thus only briefly mention those methods in which the use of SAT, or more general satisfiability modulo theories (SMT) (discussed in Chap. 11 of this Handbook [16]), is most prominent.

The basic approach described above also applies to concurrent software with interleaving semantics. In BMC for this scenario, path formulas with thread interleavings are built. Due to the potential for path explosion, numerous variants for restricting the search, path merging and compression have been considered [60, 78, 82, 143, 144]. An alternative to considering interleavings explicitly during the encoding is to build a formula in which the interleavings are encoded by means of clocks [2, 150]. Further constraint-based approaches to analyzing concurrent programs include [126, 149]. Concurrent programs can be reduced to sequential programs by applying a bound on the number of context switches [127, 143]. This transformation enables the application of analyzers for sequential programs as described above.

Verifiers for concurrent systems usually benefit from some form of partial-order reduction. Instances of BMC-based verifiers for concurrent systems that implement partial-order reduction are [70, 100, 101].

10.4.5 Bounded Model Checking for HW/SW Co-verification

The encodings described in Sect. 10.3.1 for hardware and Sect. 10.4 for software can be combined to form a single SAT instance, which enables the verification of systems that have both a hardware and a software component. This approach is the baseline for the broad area of "symbolic co-simulation" of two models, where one is written in C and the other is a hardware model in (for example) Verilog or SystemC.

A typical scenario is checking the correspondence between a "golden" hardware reference model and an RTL implementation. Another scenario is checking properties of software–hardware interaction, where the software is in C and the hardware is modeled in an HDL. There is a broad variety of styles in which ANSI-C programs or SystemC descriptions are used in these settings as (possibly partial) hardware specifications. In the special case of sequential equivalence checking between C and an HDL this is combined with heuristic insertions of *equivalence cut points*.

10.5 Encodings into Propositional SAT

In this section we elaborate on the original question of how to encode the model and the temporal specification into propositional SAT. Due to the widespread use of C to implement safety-critical software, model checking of C programs, even just for bug hunting, is an important application of formal verification. The challenge in making BMC work for a concrete programming language such as C is manyfold. First, programming languages have complex syntax and semantics which have to be parsed, analyzed and encoded. Reasoning about memory and in particular pointer arithmetic requires non-trivial decision procedures for arrays. In order to model the actual computation, including but not limited to modular arithmetic, bit-precise reasoning is indispensable.

10.5.1 Encoding Bit Vectors

At the core of SAT-based Model Checking is the encoding of word-level operations, which correspond to the evaluation of arithmetic expressions in programming languages or HDLs, into bit-level formulas. This task, also known as *bit-blasting*, is very similar to the synthesis of hardware models on the register transfer level (RTL) into net-lists. Alternatively, operations on the word-level can be modeled in the first-order *theory of bit vectors* (QF_BV).

As discussed in Chap. 11, there are various approaches to handle the bit-vector theory. Here we focus on bit-blasting. As examples we show the encoding of assignments, i.e., equality in BV, and addition of bit vectors. Other arithmetic and logical operations are treated in a similar way. Note that, in general, bit-blasting is an exponential procedure, if bit-widths, as is usually the case, are encoded logarithmically. This exponential explosion cannot be avoided, since the decision problem for full QF_BV is NEXPTIME complete [107].

After encoding models into bit vectors and bit vectors into propositional bit-level logic there remains a last step of encoding bit-level formulas into conjunctive normal form (CNF), the common input format of most SAT solvers.

In order to compactly represent formulas we need sharing. This means we use directed acyclic graphs (DAGs) or simply combinational circuits to represent generated bit-level formulas and not trees, which can be exponentially larger.

A bit-level data structure commonly used for this purpose is And-Inverter-Graphs (AIGs) [123]. AIGs are in essence representations of net-lists (Definition 2). In order to obtain a formula in CNF from an AIG it is possible to first translate the AIG into negation normal form (NNF), which at most doubles the size of the DAG, and then use the distributivity law to eliminate disjunctions over conjunctions. Each elimination of a disjunction is quadratic and thus this approach may lead to an exponential blow-up of the resulting CNF. As a consequence, translating an AIG into CNF by distribution is only feasible for small and shallow formulas. The common approach for translating formulas (and AIGs) into CNF is to use a Tseitin encoding and related optimizations, as discussed in Sect. 10.2.2.

10.5.2 Encoding Memory

Memory occurs in software but also in hardware models. The first-order *theory of arrays* is powerful enough to express most memory-related properties of practical interest. Therefore, decision procedures for the theory of arrays, as presented in Chap. 11, are essential for bounded model checking. We are mostly interested in bit-precise semantics. Thus for bounded model checking, we can focus on the quantifier-free fragment of *arrays over bit vectors* (QF_ABV).

Most of the time, memory in hardware can be handled by standard decision procedures for arrays. However, for software there are additional requirements. In particular, dynamic memory management has to be encoded.

10.5.3 Encodings with Under- and Over-approximation

The direct use of a SAT solver as cited earlier ("bit-blasting") is the conceptually simplest way to implement a bit-vector decision procedure. However, the bit-blasting approach can be too computationally expensive in practice, and there is a pressing need for better decision procedures for bit-vector arithmetic.

One frequently applied method to obtain faster decision procedures for bit-vector arithmetic and other theories is *abstraction*. The key insight is that in many cases, only a small part of the formula needs to be analyzed to conclude whether it is satisfiable or unsatisfiable. The goal of abstraction is to focus on this part of the formula.

Most decision procedures that employ abstraction implement either strict over- or under-approximations. In both cases, the desired result is a formula ϕ' that is easier to solve than the original formula ϕ.

An over-approximation of a decision problem permits more solutions than the original formula. A simple way to obtain an over-approximation for a satisfiability problem is to replace sub-formulas by new variables. In case an over-approximation ϕ' is found to be unsatisfiable, we can conclude that the original formula is unsatisfiable. Nothing, however, can be concluded if ϕ' is satisfiable, since the satisfying assignment for ϕ' need not be a satisfying assignment for ϕ.

Conversely, an under-approximation of a decision problem permits fewer solutions than the original formula. A simple way to obtain an under-approximation for a satisfiability problem is to add further constraints or to replace sub-formulas by constants. In case an under-approximation ϕ' is found to be satisfiable, we can conclude that the original formula is satisfiable. Nothing, however, can be concluded if ϕ' is unsatisfiable. A proof of unsatisfiability of ϕ' need not be a proof of unsatisfiability for ϕ.

Both over- and under-approximations can naturally be combined with forms of automated abstraction refinement, such as those pioneered in [55]. SMT solvers

implementing DPLL(T) [15, 138, 147] can be seen as performing iterative refinement (strengthening) of an over-approximation. The array theory is a very typical instance of a fragment of first-order logic that is particularly suitable for over-approximation [120, 137]. Under-approximation is frequently applied in the case of expensive bit-vector arithmetic operations such as multiplication.

In order to obtain the strengths of both over- and under-approximation, *alternation* between the two schemes can be applied. This idea is particularly fruitful if each of the two phases provides refinement information for the other. An instance of this scheme for quantifier-free Presburger arithmetic has been presented in [114]; a variant for quantifier-free bit-vector arithmetic has appeared in [41]. It is also possible to combine over- and under-approximation in a *single* abstraction, thereby forming a *mixed* abstraction. The resulting formula in general neither implies nor is implied by the original formula [38, 39].

10.6 Complete Model Checking with SAT

As explained above, the search for a counterexample of fixed length is inherently incomplete, as means to conclude the absence of counterexamples of any length are missing. We now discuss methods that enable proofs that a given property holds for unbounded depth [7].

10.6.1 Completeness Thresholds

Intuitively, if we could search *deeply enough*, we could guarantee that we have examined all the relevant behavior of the bounded program, and that searching any deeper would only exhibit states that we have explored already. A depth that provides such a guarantee is called a *completeness threshold* [119]. The notion of completeness threshold is used to determine an upper bound on the length k of counterexamples that have to be tried before the property can be declared to hold.

Computing the smallest such threshold is as hard as the model-checking problem itself, and thus, one settles in practice for over-approximations. Techniques for obtaining completeness thresholds include structural analyses of the description of the transition system [20, 21, 109], and semantic analyses of the model and the property [5, 58, 115, 119].

The completeness threshold of a design can be lowered significantly by applying abstraction techniques such as localization reduction [125]. This idea has been exploited in a number of techniques [130, 135].

10.6.2 Image Computation with SAT

BDD-based model checkers perform forward or backward fixed-point iterations in order to determine the truth of a property given in temporal logic. The key step in this

procedure is to compute a pre- or post-image of a given set of states with respect to the transition relation. Attempts have been made to emulate this fixed-point iteration using SAT solvers [1, 46, 131].

10.6.3 Basic Inductive Techniques

SAT-based techniques are well suited to check whether a given transition system satisfies a given *inductive invariant*. Recall that I denotes the initial state predicate, and that T denotes the transition relation. A state property P is *inductive* iff

1. P holds in the initial state, i.e., $I \Longrightarrow P$, and
2. P holds in all states reachable from states that satisfy P, i.e.,

$$\left(P(s) \wedge T\left(s, s'\right)\right) \Longrightarrow P\left(s'\right).$$

Observe that both conditions are quantifier-free and can therefore be checked effectively using the techniques we have described so far. The main practical problem is that a property that holds does not have to be inductive. Nothing can be concluded about P if the second condition fails. We now discuss techniques that attempt to address this case.

10.6.3.1 Strengthening the Inductive Argument

Induction can be made more likely to succeed when we check a state property P' that is stronger than the non-inductive property P. Numerous heuristics have been proposed to strengthen inductive arguments, both in the case of software and hardware models. Many initial methods relied on careful manual strengthening of properties to make them inductive, followed by automated heuristics [6].

10.6.3.2 Equivalence Reasoning

Another important preprocessing technique for bit-level model checking is based on iteratively computing the set of equivalent circuit nodes. This in particular includes the set of equivalent latches and registers. The pioneering work of van Eijk [75] consists of a greatest fixpoint computation of this equivalence relation. In essence it computes the largest equivalence relation among signals which is inductive, i.e., is preserved under the transition relation, and holds in the initial state. The resulting equivalence relation can then be used to simplify the model-checking problem by replacing equivalent nodes by representatives. An important related technique is SAT sweeping [122]. For a more complete set of references see [104].

10.6.3.3 Temporal Decomposition

Circuit nodes which are initialized to one specific constant value, true or false, and then never change, can be found in the same way. However, in many practical problems, nodes only stabilize after a certain number n of steps. In this situation, the original model-checking problem should be split into a bounded-model-checking problem for the first n steps, followed by checking a simplified model where the signals fixed after n steps are replaced by constants. This technique is called *temporal decomposition* and was introduced in [44]. Ternary simulation can be used to quickly compute an approximation of stabilizing signals.

10.6.3.4 k-Induction

An automated way to increase the strength of the inductive argument is to increase the depth of the unwinding, forming a formula that is very similar to a BMC instance. In k-induction, we first check that there is no counterexample of length k or less. We then check that no state reachable from a sequence of k-states that satisfy P violates P. Both checks can be performed effectively using a satisfiability decision procedure. The technique was first applied to hardware models [148], and then generalized to include software [64, 65]. The approach is also applicable to liveness properties, e.g., given in LTL, as ω-regular properties, or as Büchi automata [90, 91].

10.6.4 Craig Interpolation

Model checking with Craig interpolation [132] was the first robust complete SAT-based model-checking technique and is still considered to be one of the most effective techniques in practice. It uses an over-approximation of quantifier elimination, for image computation, which is obtained as an interpolation from a refutation of a BMC run between the first and the remaining states of the considered path [132]. The crucial part is an algorithm for extracting an interpolant from a resolution proof in linear time. The technique has been combined with other methods to reduce the complexity of the model, e.g., abstraction [129].

Interpolating decision procedures have been developed for numerous fragments of first-order logic, primarily with the goal of application to approximate loop invariants in program analyzers. An algorithm for interpolation in linear real arithmetic has been given in [133], for transitive relations in [156], and for full quantifier-free Presburger arithmetic in [36, 113]. An interpolating decision procedure for quantifier-free Presburger arithmetic with arrays is described in [37]. A full description of interpolation-based model checking is in Chap. 14 of this Handbook [134].

10.6.5 Iterative Inductive Strengthening

A failing inductive argument can be strengthened iteratively in a BMC-like setting, an idea exploited in the seminal algorithm IC3 [33, 34], also called *property-directed reachability checking* in [71]. As of 2013, IC3 is considered the most efficient single-engine model-checking technique for proving properties of bit-level models. In addition, it is also shown to be able to reach deep counterexamples. IC3 has been extended to full CTL, as demonstrated in [89], as well as to more general models [48, 93].

The basic idea of IC3 is to generate a relative inductive chain $F_0 \subseteq F_1 \subseteq \cdots \subseteq F_k$ of over-approximations of reachable states. "Relative inductive" means that all the successor states of F_i are in F_{i+1}. Starting with the initial state set $F_0 = I$ alone, the algorithm proceeds by either refining frontiers or by increasing k, which adds a new frontier. This process is repeated until the chain reaches a fix-point or a bad state is shown to be reachable.

The frontier sets F_i are refined by adding restrictions on states reachable in one step backward from a goal state, i.e., a bad state. These restrictions are expressed as clauses over state literals. In order to minimize their size, and speed up termination of IC3, the algorithm performs many incremental calls to a SAT solver. Initially only bad states are goal states, but after one step backward, the negation of an added clause becomes a goal too (unless the initial state is reached). These goals can thus be seen as partial models of the transition relation. Finding and minimizing these partial models is the most time-consuming part of the algorithm, and the current state of the art either uses SAT-based techniques [35] or uses ternary simulation [71].

In contrast to bounded model checking, IC3 requires many more calls to the SAT solver, typically in the range of thousands of SAT-solver calls per second. These calls, however, only check properties of one step, e.g., a single copy of the transition relation. This is a very different usage scenario for a SAT solver than in BMC. Further details and a discussion on lifting these ideas to SMT can be found in [34] or in the original publication on IC3 [33].

10.7 Abstraction Techniques Using SAT

10.7.1 Overview of Predicate Abstraction

Promoted by the success of the SLAM toolkit [8, 12, 13], *predicate abstraction* is currently the predominant abstraction technique in software model checking. Graf and Saïdi use *logical predicates* to construct an abstract domain by partitioning a program's state space [84]. The details of this procedure are described in Chap. 15 of this Handbook [99]. We focus on the use of SAT in this context.

In predicate abstraction, a sound approximation $\hat{R}$ of R is constructed using predicates over program variables. A predicate P partitions the states of a program into two classes: one in which P evaluates to true, and one in which it evaluates to false.

Each class is an *abstract state*. Let A and B be abstract states. A transition is defined from A to B (i.e., $(A, B) \in \hat{R}$) if there exists a state in A with a transition to a state in B. This construction yields an *existential abstraction* of a program, sound for reachability properties [56]. The abstract program corresponding to $\hat{R}$ is represented by a *Boolean program* [12, 13]; one with only Boolean data types, and the same control flow constructs as in C programs (including procedures). Together, n predicates partition the state space into 2^n abstract states, one for each truth assignment to all predicates.

10.7.2 Computing Abstractions with SAT

Abstractions are automatically constructed using a decision procedure to decide, for all pairs of abstract states A, B, and instructions Li, whether Li permits a transition from A to B. As n predicates lead to 2^n abstract states, this method requires $(2^n)^2$ calls to a decision procedure to compute an abstraction. In practice, a coarser but more efficiently computed *Cartesian Abstraction* (see for instance [11]) is obtained by constructing an abstraction for each predicate separately and taking the product of the resulting abstract relations.

The decision procedures are either SMT-based first-order logic theorem provers combined with theories such as machine arithmetic, for reasoning about the C programming language (e.g., ZAPATO [10] or SIMPLIFY [63]), or SAT-solvers, used to decide the satisfiability of a bit-level accurate representation of the formulas [53, 59, 120].

We now describe how an abstraction can be verified. Despite the presence of a potentially unbounded call stack, the reachability problem for sequential Boolean programs is decidable [42].[2]

The intuition is that the successor of a state is determined entirely by the top of the stack and the values of global variables, both of which take values in a finite set. Thus, for each procedure, the possible pairs of input-output values, called *summary edges*, is finite and can be cached and used during model checking [12, 79].

All existing model checkers for Boolean programs are symbolic. BDD-based tools suffer from scalability issues if the number of variables is very large. SAT-based methods scale significantly better, but cannot be used to detect fixed points. For this purpose, solvers for quantified Boolean formulas (QBF) must be used [83, 106]. However, the decision problem for QBF, a classical PSPACE-complete problem, faces the same scalability issues as BDDs. Most tools used in practice are therefore still based on BDDs, and the verification phase is often the bottleneck of predicate abstraction.

[2] In fact, all ω-regular properties are decidable for sequential Boolean programs [32].

10.7.3 Simulation with SAT

The reachability computation above may discover that an error state is reachable in the abstract program. Subsequently, a *simulation step* is used to determine whether the error exists in the concrete program or is *spurious*.

Symbolic simulation mentioned in Sect. 10.4.2, in which an abstract state is propagated through the sequence of program locations occurring in the abstract counterexample, is used to determine whether an abstract counterexample is spurious. If so, the abstraction must be *refined* to eliminate the spurious trace. This approach *does not* produce false error messages.

There are two sources of imprecision in the abstract model. *Spurious traces* arise because the set of predicates is not rich enough to distinguish between certain concrete states. *Spurious transitions* arise because the Cartesian abstraction may contain transitions not in the existential abstraction. Spurious traces are eliminated by adding additional predicates, obtained by computing the weakest precondition (or strongest postcondition) of the instructions in the trace. An alternative method is *Craig interpolation* [92]. Spurious transitions are eliminated by adding constraints to the abstract model. Such transitions are eliminated by restricting the valuations of the Boolean variables before and after the transition.

Various techniques to speed up the refinement and the simulation steps have been proposed. *Path slicing* eliminates from the counterexample instructions that do not contribute to a property violation [98]. *Loop detection* is used to compute the effect of arbitrary iterations of loops in a counterexample in a single simulation step [121]. The refinement step can be accelerated by adding statically computed invariants [22, 96], including those that eliminate a whole class of spurious counterexamples [23]. Proof-based refinement eliminates all counterexamples up to a certain length, shifting the computational effort from the verification to the refinement phase, and decreasing the number of iterations required [4].

10.7.4 Abstraction-Based Tools

The SATABS model checker uses SAT- or SMT-based abstraction, simulation and refinement [53, 54], and has also been combined with dynamic execution (testing) [86] and has been applied to concurrent software [18, 19, 157], including the scenario in which the number of threads is not bounded [102]. A proof-based technique to approximate images for bit-vector arithmetic has been proposed in [116]. Predicate abstraction has also been applied to hardware verification and HW/SW co-verification [111] and to SpecC [50] and SystemC models [29–31]. SLAM now also uses an SMT-based decision procedure [9], and experiments have been reported using a SAT-based decision procedure [59]. SAT-based checking has also been applied to the abstraction itself, i.e., to Boolean programs [17]. The LOOPFROG verifier uses SAT to compute a precise transformer for a given loop body and a given abstract domain [117, 118].

10.8 Outlook and Conclusions

We have given an overview of a broad range of SAT-based analysis techniques for both software and hardware, demonstrating the versatility of the approach.

The extension of techniques that rely on propositional SAT to the more general case of *Satisfiability Modulo Theories* (SMT) is often straightforward. The techniques described in Chap. 11 are therefore a very natural starting point for further development of the methods described here.

Early SAT-based methods have been restricted to bounded search, and are therefore typically applied for refutation, i.e., the generation of counterexamples. While bounded verification has been accepted as a useful paradigm in practical verification problems, research in recent years has extended this approach in a variety of ways to enable automated and scalable proofs for non-trivial systems.

Exciting avenues for future research include the generalization of the DPLL algorithm to rich natural domains [61] and the integration of abstraction-based methods implementing the abstract interpretation framework into SAT solvers over natural domains [66–68].

References

1. Abdulla, P.A., Bjesse, P., Eén, N.: Symbolic reachability analysis based on SAT-solvers. In: Graf, S., Schwartzbach, M.I. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 1785, pp. 411–425. Springer, Heidelberg (2000)
2. Alglave, J., Kroening, D., Tautschnig, M.: Partial orders for efficient bounded model checking of concurrent software. In: Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 8044, pp. 141–157. Springer, Heidelberg (2013)
3. Alpern, B., Wegman, M.N., Zadeck, F.K.: Detecting equality of variables in programs. In: Ferrante, J., Mager, P. (eds.) Symp. on Principles of Programming Languages (POPL), pp. 1–11. ACM, New York (1988)
4. Amla, N., McMillan, K.L.: A hybrid of counterexample-based and proof-based abstraction. In: Hu, A.J., Martin, A.K. (eds.) Formal Methods in Computer Aided Design (FMCAD). LNCS, vol. 3312, pp. 260–274. Springer, Heidelberg (2004)
5. Awedh, M., Somenzi, F.: Proving more properties with bounded model checking. In: Alur, R., Peled, D. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 3114, pp. 96–108. Springer, Heidelberg (2004)
6. Awedh, M., Somenzi, F.: Automatic invariant strengthening to prove properties in bounded model checking. In: Sentovich, E. (ed.) Design Automation Conf. (DAC), pp. 1073–1076. ACM, New York (2006)
7. Awedh, M., Somenzi, F.: Termination criteria for bounded model checking: extensions and comparison. Electron. Notes Theor. Comput. Sci. **144**(1), 51–66 (2006)
8. Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S.K., Ustuner, A.: Thorough static analysis of device drivers. In: Berbers, Y., Zwaenepoel, W. (eds.) European Conf. on Computer Systems (EuroSys), pp. 73–85. ACM, New York (2006)
9. Ball, T., Bounimova, E., Kumar, R., Levin, V.: SLAM2: Static driver verification with under 4% false alarms. In: Bloem, R., Sharygina, N. (eds.) Formal Methods in Computer Aided Design (FMCAD), pp. 35–42. IEEE, Piscataway (2010)

10. Ball, T., Cook, B., Lahiri, S.K., Zhang, L.: Zapato: automatic theorem proving for predicate abstraction refinement. In: Alur, R., Peled, D. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 3114, pp. 457–461. Springer, Heidelberg (2004)
11. Ball, T., Podelski, A., Rajamani, S.K.: Boolean and Cartesian abstraction for model checking C programs. In: Margaria, T., Yi, W. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 2031, pp. 268–283. Springer, Heidelberg (2001)
12. Ball, T., Rajamani, S.K.: Bebop: a symbolic model checker for boolean programs. In: Havelund, K., Penix, J., Visser, W. (eds.) Intl. Workshop on SPIN Model Checking and Software Verification. LNCS, vol. 1885, pp. 113–130. Springer, Heidelberg (2000)
13. Ball, T., Rajamani, S.K.: Boolean programs: a model and process for software analysis. Tech. rep., Microsoft Research (2000)
14. Barner, S., Eisner, C., Glazberg, Z., Kroening, D., Rabinovitz, I.: ExpliSAT: guiding SAT-based software verification with explicit states. In: Yorav, K. (ed.) Intl. Haifa Verification Conference (HVC). LNCS, vol. 4383, pp. 138–154. Springer, Heidelberg (2007)
15. Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 825–885. IOS Press, Amsterdam (2009)
16. Barrett, C.W., Tinelli, C.: Satisfiability modulo theories. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
17. Basler, G., Kroening, D., Weissenbacher, G.: SAT-based summarization for Boolean programs. In: Bosnacki, D., Edelkamp, S. (eds.) Intl. Workshop on Model Checking Software (SPIN). LNCS, vol. 4595, pp. 131–148 (2007)
18. Basler, G., Mazzucchi, M., Wahl, T., Kroening, D.: Symbolic counter abstraction for concurrent software. In: Bouajjani, A., Maler, O. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 5643, pp. 64–78. Springer, Heidelberg (2009)
19. Basler, G., Mazzucchi, M., Wahl, T., Kroening, D.: Context-aware counter abstraction. Form. Methods Syst. Des. **36**(3), 223–245 (2010)
20. Baumgartner, J., Kuehlmann, A.: Enhanced diameter bounding via structural transformation. In: Design, Automation & Test in Europe Conf. and Exposition (DATE), pp. 36–41. IEEE, Piscataway (2004)
21. Baumgartner, J., Kuehlmann, A., Abraham, J.A.: Property checking via structural analysis. In: Brinksma, E., Larsen, K.G. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 2404, pp. 151–165. Springer, Heidelberg (2002)
22. Beyer, D., Henzinger, T.A., Majumdar, R., Rybalchenko, A.: Invariant synthesis for combined theories. In: Cook, B., Podelski, A. (eds.) Intl. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI). LNCS, vol. 4349, pp. 378–394. Springer, Heidelberg (2007)
23. Beyer, D., Henzinger, T.A., Majumdar, R., Rybalchenko, A.: Path invariants. In: Ferrante, J., McKinley, K.S. (eds.) Conf. on Programming Language Design and Implementation (PLDI), pp. 300–309. ACM, New York (2007)
24. Biere, A.: Bounded model checking. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 457–481. IOS Press, Amsterdam (2009)
25. Biere, A., Artho, C., Schuppan, V.: Liveness checking as safety checking. In: Cleaveland, R., Garavel, H. (eds.) Intl. ERCIM Workshop on Formal Methods for Industrial Critical Systems (FMICS), pp. 160–177. Elsevier, Amsterdam (2002)
26. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, R. (ed.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
27. Biere, A., Heljanko, K., Junttila, T.A., Latvala, T., Schuppan, V.: Linear encodings of bounded LTL model checking. Log. Methods Comput. Sci. **2**(5), 1–64 (2006)

28. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press, Amsterdam (2009)
29. Blanc, N., Kroening, D.: Race analysis for SystemC using model checking. In: Intl. Conf. on Computer-Aided Design (ICCAD), pp. 356–363. IEEE, Piscataway (2008)
30. Blanc, N., Kroening, D.: Race analysis for SystemC using model checking. ACM Trans. Des. Autom. Electron. Syst. **15**(3), 1–32 (2010)
31. Blanc, N., Kroening, D., Sharygina, N.: Scoot: a tool for the analysis of SystemC models. In: Ramakrishnan, C.R., Rehof, J. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 4963, pp. 467–470. Springer, Heidelberg (2008)
32. Bouajjani, A., Esparza, J., Maler, O.: Reachability analysis of pushdown automata: application to model-checking. In: Mazurkiewicz, A.W., Winkowski, J. (eds.) Intl. Conf. on Concurrency Theory (CONCUR). LNCS, vol. 1243, pp. 135–150. Springer, Heidelberg (1997)
33. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D.A. (eds.) Intl. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI). LNCS, vol. 6538, pp. 70–87. Springer, Heidelberg (2011)
34. Bradley, A.R.: Understanding IC3. In: Cimatti, A., Sebastiani, R. (eds.) Theory and Applications of Satisfiability Testing (SAT). LNCS, vol. 7317, pp. 1–14. Springer, Heidelberg (2012)
35. Bradley, A.R., Manna, Z.: Checking safety by inductive generalization of counterexamples to induction. In: Formal Methods in Computer Aided Design (FMCAD), pp. 173–180. IEEE, Piscataway (2007)
36. Brillout, A., Kroening, D., Rümmer, P., Wahl, T.: An interpolating sequent calculus for quantifier-free Presburger arithmetic. In: Giesl, J., Hähnle, R. (eds.) Intl. Joint Conf. on Automated Reasoning (IJCAR). LNCS, vol. 6173, pp. 384–399. Springer, Heidelberg (2010)
37. Brillout, A., Kroening, D., Rümmer, P., Wahl, T.: Beyond quantifier-free interpolation in extensions of Presburger arithmetic. In: Jhala, R., Schmidt, D.A. (eds.) Intl. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI). LNCS, vol. 6538, pp. 88–102. Springer, Heidelberg (2011)
38. Brillout, A., Kroening, D., Wahl, T.: Mixed abstractions for floating-point arithmetic. In: Formal Methods in Computer Aided Design (FMCAD), pp. 69–76. IEEE, Piscataway (2009)
39. Brummayer, R., Biere, A.: Effective bit-width and under-approximation. In: Moreno-Díaz, R., Pichler, F., Quesada-Arencibia, A. (eds.) Intl. Conf. on Computer Aided Systems Theory (EUROCAST). LNCS, vol. 5717, pp. 304–311. Springer, Heidelberg (2009)
40. Bryant, R.E.: Graph Based Algorithms for Boolean Function Manipulation. Trans. Comput. **C-35**(8), 677–691 (1986)
41. Bryant, R.E., Kroening, D., Ouaknine, J., Seshia, S.A., Strichman, O., Brady, B.: Deciding bit-vector arithmetic with abstraction. In: Grumberg, O., Huth, M. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 4424, pp. 358–372. Springer, Heidelberg (2007)
42. Büchi, J.R.: Regular canonical systems. Arch. Math. Log. **6**(3–4), 91–111 (1964)
43. Cadar, C., Sen, K.: Symbolic execution for software testing: Three decades later. Commun. ACM **56**(2), 82–90 (2013)
44. Case, M.L., Mony, H., Baumgartner, J., Kanzelman, R.: Enhanced verification by temporal decomposition. In: Formal Methods in Computer Aided Design (FMCAD), pp. 17–24. IEEE, Piscataway (2009)
45. Chambers, B., Manolios, P., Vroon, D.: Faster SAT solving with better CNF generation. In: Design, Automation & Test in Europe (DATE), pp. 1590–1595. IEEE, Piscataway (2009)
46. Chauhan, P., Clarke, E.M., Kroening, D.: A SAT-based algorithm for reparameterization in symbolic simulation. In: Malik, S., Fix, L., Kahng, A.B. (eds.) Design Automation Conf. (DAC), pp. 524–529. ACM, New York (2004)
47. Chockler, H., Kroening, D., Purandare, M.: Computing mutation coverage in interpolation-based model checking. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. **31**(5), 765–778 (2012)

48. Cimatti, A., Griggio, A.: Software model checking via IC3. In: Madhusudan, P., Seshia, S.A. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 7358, pp. 277–293. Springer, Heidelberg (2012)
49. Claessen, K., Sörensson, N.: A liveness checking algorithm that counts. In: Cabodi, G., Singh, S. (eds.) Formal Methods in Computer Aided Design (FMCAD), pp. 52–59. IEEE, Piscataway (2012)
50. Clarke, E., Jain, H., Kroening, D.: Verification of SpecC using predicate abstraction. Form. Methods Syst. Des. **30**(1), 5–28 (2007)
51. Clarke, E., Kroening, D.: Hardware verification using ANSI-C programs as a reference. In: Yasuura, H. (ed.) Asia and South Pacific Design Automation Conf. (ASPDAC), pp. 308–311. IEEE, Piscataway (2003)
52. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)
53. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: Predicate abstraction of ANSI-C programs using SAT. Form. Methods Syst. Des. **25**(2–3), 105–127 (2004)
54. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: SATABS: SAT-based predicate abstraction for ANSI-C. In: Halbwachs, N., Zuck, L.D. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 3440, pp. 570–574. Springer, Heidelberg (2005)
55. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM **50**(5), 752–794 (2003)
56. Clarke, E.M., Grumberg, O., Long, D.E.: Model checking and abstraction. Trans. Program. Lang. Syst. **16**(5), 1512–1542 (1994)
57. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
58. Clarke, E.M., Kroening, D., Ouaknine, J., Strichman, O.: Computational challenges in bounded model checking. Softw. Tools Technol. Transf. **7**(2), 174–183 (2005)
59. Cook, B., Kroening, D., Sharygina, N.: Cogent: accurate theorem proving for program verification. In: Etessami, K., Rajamani, S.K. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 3576, pp. 296–300. Springer, Heidelberg (2005)
60. Cordeiro, L., Fischer, B.: Verifying multi-threaded software using SMT-based context-bounded model checking. In: Taylor, R.N., Gall, H., Medvidovic, N. (eds.) Intl. Conf. on Software Engineering (ICSE), pp. 331–340. ACM, New York (2011)
61. Cotton, S.: Natural domain SMT: a preliminary assessment. In: Chatterjee, K., Henzinger, T.A. (eds.) Intl. Conf. on Formal Modeling and Analysis of Timed Systems (FORMATS). LNCS, vol. 6246, pp. 77–91. Springer, Heidelberg (2010)
62. Currie, D.W., Hu, A.J., Rajan, S.P.: Automatic formal verification of DSP software. In: Micheli, G.D. (ed.) Design Automation Conf. (DAC), pp. 130–135. ACM, New York (2000)
63. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. Tech. rep., HP Labs (2003)
64. Donaldson, A., Haller, L., Kroening, D.: Strengthening induction-based race checking with lightweight static analysis. In: Jhala, R., Schmidt, D.A. (eds.) Intl. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI). LNCS, vol. 6538, pp. 169–183. Springer, Heidelberg (2011)
65. Donaldson, A., Kroening, D., Rümmer, P.: Automatic analysis of scratch-pad memory code for heterogeneous multicore processors. In: Esparza, J., Majumdar, R. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 6015, pp. 280–295. Springer, Heidelberg (2010)
66. D'Silva, V., Haller, L., Kroening, D.: Satisfiability solvers are static analysers. In: Miné, A., Schmidt, D. (eds.) Intl. Symp. on Static Analysis (SAS). LNCS, vol. 7460, pp. 317–333. Springer, Heidelberg (2012)
67. D'Silva, V., Haller, L., Kroening, D.: Abstract conflict driven learning. In: Giacobazzi, R., Cousot, R. (eds.) Symp. on Principles of Programming Languages (POPL), pp. 143–154. ACM, New York (2013)

68. D'Silva, V., Haller, L., Kroening, D., Tautschnig, M.: Numeric bounds analysis with conflict-driven learning. In: Flanagan, C., König, B. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 7214, pp. 48–63. Springer, Berlin (2012)
69. D'Silva, V., Kroening, D., Weissenbacher, G.: A survey of automated techniques for formal software verification. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. **27**(7), 1165–1178 (2008)
70. Dubrovin, J., Junttila, T.A., Heljanko, K.: Exploiting step semantics for efficient bounded model checking of asynchronous systems. Sci. Comput. Program. **77**(10–11), 1095–1121 (2012)
71. Eén, N., Mishchenko, A., Brayton, R.K.: Efficient implementation of property directed reachability. In: Bjesse, P., Slobodová, A. (eds.) Formal Methods in Computer Aided Design (FMCAD), pp. 125–134. FMCAD, Austin (2011)
72. Eén, N., Mishchenko, A., Sörensson, N.: Applying logic synthesis for speeding up SAT. In: Marques-Silva, J., Sakallah, K.A. (eds.) Theory and Applications of Satisfiability Testing (SAT). LNCS, vol. 4501, pp. 272–286. Springer, Heidelberg (2007)
73. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) Theory and Applications of Satisfiability Testing (SAT). LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2003)
74. Eén, N., Sterin, B., Claessen, K.: A circuit approach to LTL model checking. In: Jobstmann, B., Ray, S. (eds.) Formal Methods in Computer Aided Design (FMCAD), pp. 53–60. IEEE, Piscataway (2013)
75. van Eijk, C.A.J.: Sequential equivalence checking based on structural similarities. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. **19**(7), 814–819 (2000)
76. Eisner, C., Fisman, D.: A Practical Introduction to PSL. Springer, Heidelberg (2006)
77. Eisner, C., Fisman, D.: Functional specification of hardware via temporal logic. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
78. Emmi, M., Qadeer, S., Rakamaric, Z.: Delay-bounded scheduling. In: Ball, T., Sagiv, M. (eds.) Symp. on Principles of Programming Languages (POPL), pp. 411–422. ACM, New York (2011)
79. Finkel, A., Willems, B., Wolper, P.: A direct symbolic approach to model checking pushdown systems. Electron. Notes Theor. Comput. Sci. **9**, 27–37 (1997)
80. Gan, X., Dubrovin, J., Heljanko, K.: A symbolic model checking approach to verifying satellite onboard software. Sci. Comput. Program. **82**, 44–55 (2014)
81. Ganai, M.K., Gupta, A.: SAT-Based Scalable Formal Verification Solutions. Springer, Heidelberg (2007)
82. Ghafari, N., Hu, A.J., Rakamaric, Z.: Context-bounded translations for concurrent software: an empirical evaluation. In: van de Pol, J., Weber, M. (eds.) Intl. Workshop on Model Checking Software (SPIN). LNCS, vol. 6349, pp. 227–244. Springer, Heidelberg (2010)
83. Giunchiglia, E., Marin, P., Narizzano, M.: Reasoning with quantified boolean formulas. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 761–780. IOS Press, Amsterdam (2009)
84. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
85. Groce, A., Kroening, D.: Making the most of BMC counterexamples. Electron. Notes Theor. Comput. Sci. **119**, 67–81 (2005)

86. Groce, A., Kroening, D., Clarke, E.: Counterexample guided abstraction refinement via program execution. In: Davies, J., Schulte, W., Barnett, M. (eds.) Intl. Conf. on Formal Engineering Methods (ICFEM). LNCS, vol. 3308, pp. 224–238. Springer, Heidelberg (2004)
87. Groce, A., Kroening, D., Lerda, F.: Understanding counterexamples with explain. In: Alur, R., Peled, D.A. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 3114, pp. 453–456. Springer, Heidelberg (2004)
88. Gupta, A., Kahlon, V., Qadeer, S., Touili, T.: Model checking concurrent programs. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
89. Hassan, Z., Bradley, A.R., Somenzi, F.: Incremental, inductive CTL model checking. In: Madhusudan, P., Seshia, S.A. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 7358, pp. 532–547. Springer, Heidelberg (2012)
90. Heljanko, K., Junttila, T.A., Keinänen, M., Lange, M., Latvala, T.: Bounded model checking for weak alternating Büchi automata. In: Ball, T., Jones, R.B. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 4144, pp. 95–108. Springer, Heidelberg (2006)
91. Heljanko, K., Junttila, T.A., Latvala, T.: Incremental and complete bounded model checking for full PLTL. In: Etessami, K., Rajamani, S.K. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 3576, pp. 98–111. Springer, Heidelberg (2005)
92. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: Jones, N.D., Leroy, X. (eds.) Symp. on Principles of Programming Languages (POPL), pp. 232–244. ACM, New York (2004)
93. Hoder, K., Bjørner, N.: Generalized property directed reachability. In: Cimatti, A., Sebastiani, R. (eds.) Theory and Applications of Satisfiability Testing (SAT). LNCS, vol. 7317, pp. 157–171. Springer, Heidelberg (2012)
94. Ivancic, F., Shlyakhter, I., Gupta, A., Ganai, M.K.: Model checking C programs using F-SOFT. In: Intl. Conf. on Computer Design (ICCD), pp. 297–308. IEEE, Piscataway (2005)
95. Jain, H., Clarke, E.M.: Efficient SAT solving for non-clausal formulas using DPLL, graphs, and watched cuts. In: Design Automation Conf. (DAC), pp. 563–568. ACM, New York (2009)
96. Jain, H., Ivancic, F., Gupta, A., Shlyakhter, I., Wang, C.: Using statically computed invariants inside the predicate abstraction and refinement loop. In: Ball, T., Jones, R.B. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 4144, pp. 137–151. Springer, Heidelberg (2006)
97. Järvisalo, M., Biere, A., Heule, M.: Simulating circuit-level simplifications on CNF. J. Autom. Reason. **49**(4), 583–619 (2012)
98. Jhala, R., Majumdar, R.: Path slicing. In: Sarkar, V., Hall, M.W. (eds.) Conf. on Programming Language Design and Implementation (PLDI), pp. 38–47. ACM, New York (2005)
99. Jhala, R., Podelski, A., Rybalchenko, A.: Predicate abstraction for program verification. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
100. Jussila, T., Heljanko, K., Niemelä, I.: BMC via on-the-fly determinization. Electron. Notes Theor. Comput. Sci. **89**(4), 561–577 (2003)
101. Kahlon, V., Wang, C., Gupta, A.: Monotonic partial order reduction: an optimal symbolic partial order reduction technique. In: Bouajjani, A., Maler, O. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 5643, pp. 398–413. Springer, Heidelberg (2009)
102. Kaiser, A., Kroening, D., Wahl, T.: Dynamic cutoff detection in parameterized concurrent programs. In: Touili, T., Cook, B., Jackson, P. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 6174, pp. 654–659. Springer, Heidelberg (2010)
103. Kautz, H.A., Selman, B.: Pushing the envelope: planning, propositional logic and stochastic search. In: Clancey, W.J., Weld, D.S. (eds.) National Conf. on Artificial Intelligence (AAAI), pp. 1194–1201. AAAI Press/MIT Press, Portland/Cambridge (1996)
104. Khasidashvili, Z., Nadel, A.: Implicative simultaneous satisfiability and applications. In: Eder, K., Lourenço, J., Shehory, O. (eds.) Intl. Haifa Verification Conference (HVC). LNCS, vol. 7261, pp. 66–79. Springer, Heidelberg (2011)

105. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976)
106. Kleine Büning, H., Bubeck, U.: Theory of quantified boolean formulas. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 735–760. IOS Press, Amsterdam (2009)
107. Kovásznai, G., Fröhlich, A., Biere, A.: On the complexity of fixed-size bit-vector logics with binary encoded bit-width. In: Fontaine, P., Goel, A. (eds.) Intl. Workshop on Satisfiability Modulo Theories (SMT), pp. 44–55 (2012). EasyChair
108. Kroening, D.: Application specific higher order logic theorem proving. In: Autexier, S., Mantel, H. (eds.) Proc. of the Verification Workshop (VERIFY), pp. 5–15 (2002)
109. Kroening, D.: Computing over-approximations with bounded model checking. Electron. Notes Theor. Comput. Sci. **144**(1), 79–92 (2006)
110. Kroening, D.: Software verification. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 505–532. IOS Press, Amsterdam (2009)
111. Kroening, D., Clarke, E.: Checking consistency of C and Verilog using predicate abstraction and induction. In: Intl. Conf. on Computer-Aided Design (ICCAD), pp. 66–72. IEEE/ACM, Piscataway/New York (2004)
112. Kroening, D., Clarke, E., Yorav, K.: Behavioral consistency of C and Verilog programs using bounded model checking. In: Design Automation Conf. (DAC), pp. 368–371. ACM, New York (2003)
113. Kroening, D., Leroux, J., Rümmer, P.: Interpolating quantifier-free Presburger arithmetic. In: Fermüller, C.G., Voronkov, A. (eds.) Intl. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR). LNCS, vol. 6397, pp. 489–503. Springer, Heidelberg (2010)
114. Kroening, D., Ouaknine, J., Seshia, S., Strichman, O.: Abstraction-based satisfiability solving of Presburger arithmetic. In: Alur, R., Peled, D.A. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 3114, pp. 308–320. Springer, Heidelberg (2004)
115. Kroening, D., Ouaknine, J., Strichman, O., Wahl, T., Worrell, J.: Linear completeness thresholds for bounded model checking. In: Gopalakrishnan, G., Qadeer, S. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 6806, pp. 557–572. Springer, Heidelberg (2011)
116. Kroening, D., Sharygina, N.: Approximating predicate images for bit-vector logic. In: Hermanns, H., Palsberg, J. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 3920, pp. 242–256. Springer, Heidelberg (2006)
117. Kroening, D., Sharygina, N., Tonetta, S., Tsitovich, A., Wintersteiger, C.: Loopfrog: a static analyzer for ANSI-C programs. In: Intl. Conf. on Automated Software Engineering (ASE), pp. 668–670. IEEE, Piscataway (2009)
118. Kroening, D., Sharygina, N., Tonetta, S., Tsitovich, A., Wintersteiger, C.M.: Loop summarization using abstract transformers. In: Cha, S.D., Choi, J., Kim, M., Lee, I., Viswanathan, M. (eds.) Intl. Symp. on Automated Technology for Verification and Analysis (ATVA). LNCS, vol. 5311, pp. 111–125. Springer, Heidelberg (2008)
119. Kroening, D., Strichman, O.: Efficient computation of recurrence diameters. In: Zuck, L.D., Attie, P.C., Cortesi, A., Mukhopadhyay, S. (eds.) Intl. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI). LNCS, vol. 2575, pp. 298–309. Springer, Heidelberg (2003)
120. Kroening, D., Strichman, O.: Decision Procedures. Springer, Heidelberg (2008)
121. Kroening, D., Weissenbacher, G.: Counterexamples with loops for predicate abstraction. In: Ball, T., Jones, R.B. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 4144, pp. 152–165. Springer, Heidelberg (2006)
122. Kuehlmann, A.: Dynamic transition relation simplification for bounded property checking. In: Intl. Conf. on Computer-Aided Design (ICCAD), pp. 50–57. IEEE/ACM, Piscataway/New York (2004)

123. Kuehlmann, A., Paruthi, V., Krohm, F., Ganai, M.K.: Robust boolean reasoning for equivalence checking and functional property verification. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. **21**(12), 1377–1394 (2002)
124. Kuismin, T., Heljanko, K.: Increasing confidence in liveness model checking results with proofs. In: Bertacco, V., Legay, A. (eds.) Intl. Haifa Verification Conference (HVC). LNCS, vol. 8244, pp. 32–43. Springer, Heidelberg (2013)
125. Kurshan, R.P.: Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach. Princeton University Press, Princeton (1994)
126. Lahiri, S.K., Qadeer, S., Rakamaric, Z.: Static and precise detection of concurrency errors in systems code using SMT solvers. In: Bouajjani, A., Maler, O. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 5643, pp. 509–524. Springer, Heidelberg (2009)
127. Lal, A., Reps, T.W.: Reducing concurrent analysis under a context bound to sequential analysis. Form. Methods Syst. Des. **35**(1), 73–97 (2009)
128. Latvala, T., Biere, A., Heljanko, K., Junttila, T.A.: Simple bounded LTL model checking. In: Hu, A.J., Martin, A.K. (eds.) Formal Methods in Computer Aided Design (FMCAD). LNCS, vol. 3312, pp. 186–200. Springer, Heidelberg (2004)
129. Li, B., Somenzi, F.: Efficient abstraction refinement in interpolation-based unbounded model checking. In: Hermanns, H., Palsberg, J. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 3920, pp. 227–241. Springer, Heidelberg (2006)
130. Li, B., Wang, C., Somenzi, F.: Abstraction refinement in symbolic model checking using satisfiability as the only decision procedure. Softw. Tools Technol. Transf. **7**(2), 143–155 (2005)
131. McMillan, K.L.: Applying SAT methods in unbounded symbolic model checking. In: Brinksma, E., Larsen, K.G. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 2404, pp. 250–264. Springer, Heidelberg (2002)
132. McMillan, K.L.: Interpolation and SAT-based model checking. In: Hunt, W.A. Jr., Somenzi, F. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 2725, pp. 1–13. Springer, Heidelberg (2003)
133. McMillan, K.L.: An interpolating theorem prover. In: Jensen, K., Podelski, A. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 2988, pp. 16–30. Springer, Heidelberg (2004)
134. McMillan, K.L.: Interpolation and model checking. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
135. McMillan, K.L., Amla, N.: Automatic abstraction without counterexamples. In: Garavel, H., Hatcliff, J. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 2619, pp. 2–17. Springer, Heidelberg (2003)
136. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient SAT solver. In: Design Automation Conf. (DAC), pp. 530–535. ACM, New York (2001)
137. de Moura, L.M., Bjørner, N.: Generalized, efficient array decision procedures. In: Formal Methods in Computer Aided Design (FMCAD), pp. 45–52. IEEE, Piscataway (2009)
138. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). J. ACM **53**(6), 937–977 (2006)
139. Penczek, W., Wozna, B., Zbrzezny, A.: Bounded model checking for the universal fragment of CTL. Fundam. Inform. **51**(1–2), 135–156 (2002)
140. Piterman, N., Pnueli, A.: Temporal logic and fair discrete systems. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
141. Prasad, M., Biere, A., Gupta, A.: A survey on recent advances in SAT-based formal verification. Softw. Tools Technol. Transf. **7**(2), 156–173 (2005)

142. Prestwich, S.D.: CNF encodings. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 75–97. IOS Press, Amsterdam (2009)
143. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: Halbwachs, N., Zuck, L.D. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 3440, pp. 93–107. Springer, Heidelberg (2005)
144. Rabinovitz, I., Grumberg, O.: Bounded model checking of concurrent programs. In: Etessami, K., Rajamani, S.K. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 3576, pp. 82–97. Springer, Heidelberg (2005)
145. Rintanen, J.: Planning as satisfiability: heuristics. Artif. Intell. **193**, 45–86 (2012)
146. Schuppan, V., Biere, A.: Efficient reduction of finite state model checking to reachability analysis. Softw. Tools Technol. Transf. **5**(1–2), 185–204 (2004)
147. Sebastiani, R.: Lazy satisfiability modulo theories. J. Satisf. Boolean Model. Comput. **3**(3–4), 141–224 (2007)
148. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In: Hunt, W.A. Jr., Johnson, S.D. (eds.) Formal Methods in Computer Aided Design (FMCAD). LNCS, vol. 1954, pp. 108–125. Springer, Heidelberg (2000)
149. Sinha, N., Wang, C.: Staged concurrent program analysis. In: Roman, G.C., Sullivan, K.J. (eds.) Intl. Symp. on Foundations of Software Engineering (FSE), pp. 47–56. ACM, New York (2010)
150. Sinha, N., Wang, C.: On interference abstractions. In: Ball, T., Sagiv, M. (eds.) Symp. on Principles of Programming Languages (POPL), pp. 423–434. ACM, New York (2011)
151. Tseitin, G.S.: On the complexity of derivation in propositional calculus. In: Studies in Constructive Mathematics and Mathematical Logic, Part II. Seminars in Mathematics, vol. 8, pp. 234–259 (1968). V.A. Steklov Mathematical Institute. English Translation, Consultants Bureau, pp. 115–125 (1970)
152. Vardi, M.Y., Wolper, P.: Reasoning about infinite computations. Inf. Comput. **115**(1), 1–37 (1994)
153. Velev, M.N.: Efficient translation of boolean formulas to CNF in formal verification of microprocessors. In: Imai, M. (ed.) Asia and South Pacific Design Automation Conf. (ASPDAC), pp. 310–315. IEEE, Piscataway (2004)
154. Vijayaraghavan, S., Ramanathan, M.: A Practical Guide for SystemVerilog Assertions. Springer, Heidelberg (2005)
155. Vizel, Y., Weissenbacher, G., Malik, S.: Boolean satisfiability solvers and their applications in model checking. Proc. IEEE **103**(11), 2021–2035 (2015). doi:10.1109/JPROC.2015.2455034
156. Weissenbacher, G., Kroening, D.: An interpolating decision procedure for transitive relations with uninterpreted functions. In: Namjoshi, K.S., Zeller, A., Ziv, A. (eds.) Intl. Haifa Verification Conference (HVC). LNCS, vol. 6405, pp. 150–168. Springer, Heidelberg (2009)
157. Witkowski, T., Blanc, N., Weissenbacher, G., Kroening, D.: Model checking concurrent Linux device drivers. In: Stirewalt, R.E.K., Egyed, A., Fischer, B. (eds.) Intl. Conf. on Automated Software Engineering (ASE), pp. 501–504. ACM, New York (2007)

Chapter 11
Satisfiability Modulo Theories

Clark Barrett and Cesare Tinelli

Abstract Satisfiability Modulo Theories (SMT) refers to the problem of determining whether a first-order formula is satisfiable with respect to some logical theory. Solvers based on SMT are used as back-end engines in model-checking applications such as bounded, interpolation-based, and predicate-abstraction-based model checking. After a brief illustration of these uses, we survey the predominant techniques for solving SMT problems with an emphasis on the *lazy* approach, in which a propositional satisfiability (SAT) solver is combined with one or more *theory solvers*. We discuss the architecture of a lazy SMT solver, give examples of theory solvers, show how to combine such solvers modularly, and mention several extensions of the lazy approach. We also briefly describe the *eager* approach in which the SMT problem is reduced to a SAT problem. Finally, we discuss how the basic framework for determining satisfiability can be extended with additional functionality such as producing models, proofs, unsatisfiable cores, and interpolants.

11.1 Introduction

In several areas of computer science, including formal verification of hardware and software, many important problems can be reduced to checking the satisfiability of a formula in some logic. Several of these problems can be naturally formulated as satisfiability problems in propositional logic and solved very efficiently by modern SAT solvers, as described in Chap. 10 of this book [23]. Other problems are formulated more naturally and compactly in classical logics, such as first-order or higher-order logics, with a more expressive language that includes non-Boolean variables, function and predicate symbols (with positive arity) and quantifiers. There is, of course, a trade-off between the expressiveness of a logic and the ability to automatically check the satisfiability of its formulas.

C. Barrett
Stanford University, Stanford, CA, USA

C. Tinelli (✉)
The University of Iowa, Iowa City, IA, USA
e-mail: cesare-tinelli@uiowa.edu

E.M. Clarke et al. (eds.), *Handbook of Model Checking*,
DOI 10.1007/978-3-319-10575-8_11

A practical compromise can be achieved with fragments of first-order logic that are restricted either syntactically, for instance by allowing only certain classes of formulas, or semantically, by constraining the interpretation of certain function and predicate symbols, or both. Such restrictions can make the satisfiability problem decidable and, more importantly, allow the development of specialized satisfiability procedures that exploit properties of the fragment to great advantage for practical efficiency, even in cases with high worst-case computational complexity. When semantic restrictions are involved, they can be understood as limiting the interpretations of certain symbols to models of some logical *background theory* (e.g., the theory of equality, of integer numbers, of real numbers, of arrays, of lists, and so on). In such cases, we speak of *Satisfiability Modulo Theories* (*SMT*).[1]

Building on classical results on decision procedures for first-order reasoning, and on the tremendous advances in SAT-solving technology in the last two decades, SMT has grown in recent years into a very active research field whose defining feature is the use of reasoning methods specific to logical theories of interest in target applications. Thanks to advances in SMT research and technology, there are now several powerful and sophisticated SMT solvers (e.g., Alt-Ergo [26], Beaver [94], Boolector [36], CVC4 [10], MathSAT5 [53], openSMT [39], SMTInterpol [51], SONOLAR [135], STP [76], veriT [29], Yices [69], and Z3 [121]) which are being used in a rapidly expanding set of applications. Application areas currently include processor verification, equivalence checking, bounded and unbounded model checking, predicate abstraction, static analysis, automated test case generation, extended static checking, type checking, planning, scheduling, and optimization.

The recent progress in SMT has been driven by several factors, including: a focus on background theories and classes of problems that occur in practice; liftings and adaptations of SAT technology to the SMT case; innovations in core algorithms and data structures; development of abstract constraint-solving frameworks and general solver architectures to guide efficient implementations; novel search heuristics; and attention to implementation details.[2] A major enabler of this progress has been SMT-LIB [14], a standardization and benchmark collection initiative collectively developed and supported by the SMT community, together with its derivative activities: the SMT workshop, an international forum for SMT researchers and users of SMT applications or techniques; SMT-COMP [20], an international competition for SMT solvers supporting the SMT-LIB input/output format [15]; and SMT-EXEC, a public execution service allowing researchers to run experimental evaluations on SMT solvers.[3]

This chapter provides a fairly high-level overview of SMT and its main results and techniques, together with references to the relevant literature for a deeper study. It concentrates mostly on the predominant approach for implementing SMT solvers, known as the "lazy approach," wherein an efficient and properly instrumented SAT

[1] This terminology originated in [156] and was popularized by the SMT-LIB initiative [14].

[2] It is worth noting that many of the same factors are driving improvements in modern SAT research (see [22] as well as Chap. 9 of this book).

[3] http://smtlib.org, http://smt-workshop.org, http://www.smtexec.org.

solver is combined with one or more *theory solvers*, highly specialized solvers for problems consisting just of conjunctions of *theory literals*—atomic and negated atomic formulas in the language of some particular theory T.

The chapter is structured as follows. The rest of this section provides technical background information, defining basic notions and terminology used throughout the chapter. Section 11.2 gives an overview of some uses of SMT solvers in model checking applications. Section 11.3 describes the lazy approach to SMT in which a SAT solver and a theory solver cooperate to solve an SMT problem. Section 11.4 discusses theory solvers for a number of background theories used in SMT applications, and specifically in model checking. Section 11.5 focuses on techniques for combining theory solvers for different theories into a solver for a combination of those theories. Section 11.6 discusses a few extensions and enhancements to the lazy approach. Section 11.7 describes an alternative to the lazy approach for SMT, aptly named the "eager approach," which takes advantage of SAT solvers more directly. Finally, Sect. 11.8 presents a number of important functionalities provided by modern SMT solvers that go beyond mere satisfiability checking, and that have been crucial to the success of SMT as an enabling technology in applications like model checking.

11.1.1 Technical Preliminaries

SMT problems are formulated within first-order logic with equality. Since many applications of SMT involve different data types, it is more convenient to work with a sorted (i.e., typed) version of that logic, as opposed to the classical unsorted version. In this chapter we use a basic version of *many-sorted* logic [71, 114], which is adequate for our purposes. More sophisticated typed logics are sometimes used in the literature. For instance, the SMT-LIB 2 standard is based on a sorted logic with non-nullary sort symbols and *let* binders [16]. Other work adopts, and advocates for, a first-order logic with parametric (universal) types [26, 108, 109].

Syntax We fix an infinite set $\mathbf{S}$ of *sort symbols* and consider an infinite set $\mathbf{X}$ of *(sorted) variables*, each uniquely associated with a sort in $\mathbf{S}$. A many-sorted *signature* Σ consists of a set $\Sigma^{\mathrm{S}} \subseteq S$ of sort symbols; a set Σ^{P} of *predicate symbols*; a set Σ^{F} of *function symbols*; a total mapping from Σ^{P} to the set $(\Sigma^{\mathrm{S}})^*$ of strings over Σ^{S}; and a total mapping from Σ^{F} to the set $(\Sigma^{\mathrm{S}})^+$ of non-empty strings over Σ^{S}—where * and $^+$ are the usual regular expression operators. For $n \geq 0$, a function symbol f (resp., predicate symbol p) has a unique[4] *arity* n and *rank* $\sigma_1 \cdots \sigma_n \sigma$ (resp., $\sigma_1 \cdots \sigma_n$) in Σ if it is mapped to the sort sequence $\sigma_1 \cdots \sigma_n \sigma$ (resp., $\sigma_1 \cdots \sigma_n$). When n above is 0, f is also called a *constant symbol (of sort σ)* and p a *propositional symbol*. A signature Σ is a *subsignature* of a signature Ω, written $\Sigma \subseteq \Omega$, and Ω is a *supersignature* of Σ, if $\Sigma^{\mathrm{S}} \subseteq \Omega^{\mathrm{S}}$, $\Sigma^{\mathrm{F}} \subseteq \Omega^{\mathrm{F}}$, $\Sigma^{\mathrm{P}} \subseteq \Omega^{\mathrm{P}}$, and every function or predicate symbol of Σ has the same rank in Σ as in Ω.

[4] For simplicity, we do not allow any form of symbol overloading here.

A *(Σ-)term of sort σ* is either a sorted variable x of sort $\sigma \in \Sigma^{\mathrm{S}}$ or an expression of the form $f(t_1, \ldots, t_n)$ with $n \geq 0$ where $f \in \Sigma^{\mathrm{F}}$ with rank $\sigma_1 \cdots \sigma_n \sigma$ and t_i is a term of sort σ_i for $i = 1, \ldots, n$. An *atomic (Σ-)formula* is either the symbol $\bot$, for falsity; an expression of the form $t_1 = t_2$ with t_1, t_2 terms of the same sort;[5] or an expression of the form $p(t_1, \ldots, t_n)$ with $n \geq 0$ where $p \in \Sigma^{\mathrm{P}}$ with rank $\sigma_1 \cdots \sigma_n$ and t_i is a Σ-term of sort σ_i for $i = 1, \ldots, n$. A *(Σ-)literal* is an atomic Σ-formula or an expression $\neg\varphi$ where φ is an atomic Σ-formula. A *(Σ-)formula* is an atomic Σ-formula or an expression of the form $\neg\varphi$, $\varphi \vee \psi$, or $\exists x\, \varphi$ where x is a variable with sort in Σ^{S} and φ and ψ are Σ-formulas. We will write $\exists x{:}\sigma\, \varphi$ instead of $\exists x\, \varphi$ to indicate that x has sort σ. The other logical connectives as well as the universal quantifier can be formally defined in terms of the logical symbols above as usual (e.g., $\varphi \Rightarrow \psi$ as a shorthand for $\neg\varphi \vee \psi$; $\forall x\, \varphi$ as a shorthand for $\neg\exists x\, \neg\varphi$; and so on). Examples of signatures and formulas used in SMT are provided in Sect. 11.4.

Free occurrences of a variable in a formula are defined as usual: all variable occurrences in atomic formulas are free; a variable x distinct from a variable y occurs free in a formula $\neg\varphi$, $\varphi_1 \vee \varphi_2$, or $\exists y.\psi$ iff it occurs free respectively in φ, in φ_1 or φ_2, or in ψ. A *(Σ-)sentence* is a Σ-formula with no free variables. If φ is a Σ-formula and $\mathbf{x} = (x_1, \ldots, x_n)$ a tuple of distinct variables, we will write $\varphi[\mathbf{x}]$ or $\varphi[x_1, \ldots, x_n]$ to express that the free variables of φ are in $\mathbf{x}$; furthermore, if $t_1, \ldots, t_n$ are terms with each t_i of the same sort as x_i, we will write $\varphi[t_1, \ldots, t_n]$ to denote the formula obtained from $\varphi[x_1, \ldots, x_n]$ by simultaneously replacing each occurrence of x_i in φ by t_i, for $i = 1, \ldots, n$.

Semantics For each signature Σ and set $X \subseteq \mathbf{X}$ of variables whose sorts are in Σ^{S}, a *Σ-interpretation $\mathscr{A}$ over X* maps

- each sort $\sigma \in \Sigma^{\mathrm{S}}$ to a non-empty set A_σ, the *domain* of σ in $\mathscr{A}$;
- each variable $x \in X$ of sort σ to an element $x^{\mathscr{A}} \in A_\sigma$;
- each function symbol $f \in \Sigma^{\mathrm{F}}$ of rank $\sigma_1 \cdots \sigma_n \sigma$ to a total function $f^{\mathscr{A}} : A_{\sigma_1} \times \cdots \times A_{\sigma_n} \to A_\sigma$ (and in particular each constant c of sort σ to a $c^{\mathscr{A}} \in A_\sigma$),
- each predicate symbol $p \in \Sigma^{\mathrm{P}}$ of rank $\sigma_1 \cdots \sigma_n$ to a relation $p^{\mathscr{A}} \subseteq A_{\sigma_1} \times \cdots \times A_{\sigma_n}$.

A *Σ-model* is a Σ-interpretation over an empty set of variables. Let $\mathscr{A}$ be an Ω-interpretation over some set Y of variables. When $\Sigma \subseteq \Omega$ and $X \subseteq Y$, we denote by $\mathscr{A}^{\Sigma,X}$ the *reduct* of $\mathscr{A}$ to (Σ, X), i.e., the Σ-interpretation over X obtained from $\mathscr{A}$ by restricting it to interpret only the symbols in Σ and the variables in X. $\mathscr{A}$ is an *expansion* of a Σ-interpretation $\mathscr{B}$ over X if $\mathscr{B} = \mathscr{A}^{\Sigma,X}$.

Every Σ-interpretation $\mathscr{A}$ over some $X \subseteq \mathbf{X}$ induces a unique mapping $(\_)^{\mathscr{A}}$ from Σ-terms $f(t_1, \ldots, t_n)$ with variables in X to elements of sort domains such that $(f(t_1, \ldots, t_n))^{\mathscr{A}} = f^{\mathscr{A}}(t_1^{\mathscr{A}}, \ldots, t_n^{\mathscr{A}})$. We define a satisfiability relation $\models$ between such interpretations and Σ-formulas with variables in X inductively as fol-

[5] We will use '=' also to denote equality at the meta-level, relying on the context for disambiguation.

lows:

$$
\begin{array}{lll}
\mathscr{A} \not\models \bot & & \\
\mathscr{A} \models t_1 = t_2 & \text{iff} & t_1^{\mathscr{A}} = t_n^{\mathscr{A}} \\
\mathscr{A} \models p(t_1, \ldots, t_n) & \text{iff} & (t_1^{\mathscr{A}}, \ldots, t_n^{\mathscr{A}}) \in p^{\mathscr{A}} \\
\mathscr{A} \models \neg\varphi & \text{iff} & \mathscr{A} \not\models \varphi \\
\mathscr{A} \models \varphi \vee \psi & \text{iff} & \mathscr{A} \models \varphi \text{ or } \mathscr{A} \models \psi \\
\mathscr{A} \models \exists x{:}\sigma\, \varphi & \text{iff} & \mathscr{A}[x \mapsto a] \models \varphi \text{ for some } a \in \mathscr{A}_\sigma
\end{array}
$$

where $\mathscr{A}[x \mapsto a]$ denotes the Σ-interpretation that maps x to a and is otherwise identical to $\mathscr{A}$. A Σ-interpretation $\mathscr{A}$ *satisfies* a Σ-formula φ if $\mathscr{A} \models \varphi$. A set Φ of Σ-formulas *entails* a Σ-formula φ, written $\Phi \models \varphi$, iff every Σ-interpretation that satisfies all formulas in Φ satisfies φ as well. The set Φ is *satisfiable* iff $\Phi \not\models \bot$, and φ is *valid* iff it is entailed by the empty set.

Theories In SMT, one is not interested in arbitrary models but in models belonging to a given *theory* T constraining the interpretation of the symbols in some signature Σ. We define theories most generally as classes of models with the same signature. More precisely, a *Σ-theory* T is a pair $(\Sigma, \mathbf{A})$ where Σ is a signature and $\mathbf{A}$ is a class (in the sense of set theory) of Σ-models. Section 11.4 discusses several examples of theories commonly used in SMT.

Let $T = (\Sigma, \mathbf{A})$ be a Σ-theory. A *T-interpretation* is any Ω-interpretation $\mathscr{A}$ for some $\Omega \supseteq \Sigma$ such that $\mathscr{A}^{\Sigma,\emptyset} \in \mathbf{A}$. A formula φ is *satisfiable in T*, or *T-satisfiable*, if it is satisfied by some T-interpretation $\mathscr{A}$.[6] A set Φ of Ω-formulas *T-entails* an Ω-formula φ, written $\Phi \models_T \varphi$, iff every T-interpretation that satisfies all formulas in Φ satisfies φ as well. The set Φ is *T-satisfiable* iff $\Phi \not\models_T \bot$, and φ is *T-valid* iff φ is T-entailed by the empty set, written as $\models_T \varphi$. *T-unsatisfiable* abbreviates not T-satisfiable. These notions reduce to the corresponding ones given earlier when T is the class of all Σ-models.

Note that, as defined here, T-interpretations allow us to consider the satisfiability in a Σ-theory T of formulas that contain sort, predicate or function symbols not in Σ. These symbols are traditionally called *uninterpreted*. In SMT applications, it is convenient to use formulas with uninterpreted constant symbols, which for satisfiability purposes are analogous to free variables, or with uninterpreted predicate/function symbols, which can be used as abstractions of other formulas/terms or of operators not in the theory.

Also note that the notions of theory and T-validity presented here are more general than those used traditionally in first-order theorem proving, where a theory is defined as a recursive set of sentences, the *axioms* of the theory, and T-validity is defined as entailment by those axioms. The reason is that every set A of Σ-sentences is characterized by (i.e., has the same set of valid sentences as) a class of Σ-models,

[6] Observe that the class of all T-interpretations includes all possible expansions of models in $\mathbf{A}$. This essentially means that variables and sort, function, and predicate symbols not in Σ can be interpreted arbitrarily.

namely the Σ-models of A. In contrast, not every class of Σ-models is characterized by a recursive, or even non-recursive, set of (first-order) axioms.[7]

In the next sections, we will often consider the T-satisfiability of conjunctions (or, equivalently, sets) of literals. We will refer to these conjunctions as *constraints* and talk about *constraint satisfiability* in T.

Abstractions SMT techniques often use propositional abstractions of first-order sentences. Since our logic properly embeds propositional logic, such abstractions can be defined as follows. Let us fix a signature Π consisting exclusively of an infinite set of propositional symbols not contained in any theory signature. Every quantifier-free formula (*qff*) of signature Π is in effect a propositional formula, satisfiable in our sense iff it is satisfiable in propositional logic. For every theory signature Σ, we define an injective mapping $(\_)^{\mathrm{a}}$ from the set of all atomic Σ-formulas into Π. This mapping extends homomorphically to an (injective) mapping, also denoted as $(\_)^{\mathrm{a}}$, from quantifier-free Σ-formulas to quantifier-free Π-formulas (i.e., propositional formulas) such that $(\neg\varphi)^{\mathrm{a}} = \neg(\varphi^{\mathrm{a}})$ and $(\varphi \vee \psi)^{\mathrm{a}} = \varphi^{\mathrm{a}} \vee \psi^{\mathrm{a}}$ for all qffs φ and ψ. We denote by $(\_)^{\mathrm{c}}$ the inverse homomorphism of $(\_)^{\mathrm{a}}$, which is such that $(\varphi^{\mathrm{a}})^{\mathrm{c}} = \varphi$, $(\neg\varphi)^{\mathrm{c}} = \neg(\varphi^{\mathrm{c}})$, and $(\varphi \vee \psi)^{\mathrm{c}} = \varphi^{\mathrm{c}} \vee \psi^{\mathrm{c}}$ for all qffs φ and ψ.

We call an *SMT solver* any program that tries to determine the satisfiability of some class **C** of formulas in some theory T. What distinguishes SMT as a field is the development and use of efficient reasoning techniques specific to the selected theory and class of formulas.

11.2 SMT in Model Checking

Model checking has leveraged SMT extensively in the last decade thanks to the impressive growth in the performance and scope of SMT solvers. The use of SMT solving in support of software model checking and, more generally, model checking for infinite-state transition systems is now widespread, as can be seen in the rest of this book. In this section, we give a general—and necessarily incomplete—sampling of that by focusing on a few major model-checking methods.

Roughly speaking, we could say that all of these methods rely on some encoding of a software or hardware system under analysis as a transition system S whose state space is represented by the Cartesian product $D_1 \times \cdots \times D_n$ of finite or infinite domains (Booleans, fixed-size bit vectors, integers, and so on) modeled by some Σ-theory T. The system itself is (implicitly or explicitly) described by a pair $(I[\mathbf{x}], Tr[\mathbf{x}, \mathbf{x}'])$ of typically quantifier-free Σ-formulas[8] where

[7] A well-known example of the latter would the SMT theory consisting of a single Σ-interpretation for the integers with the usual operations.

[8] This is an oversimplification because, for instance, several software model-checking methods also rely for efficiency on a separate representation of a program's control structure as a control flow graph. See Chap. 15 for more details [96].

- $\mathbf{x}$ and $\mathbf{x}'$ are n-tuples of variables semantically ranging over $D_1 \times \cdots \times D_n$;
- $I[\mathbf{x}]$ is satisfied exactly by the initial states of S;
- $Tr[\mathbf{x}, \mathbf{x}']$ is satisfied by all pairs $\mathbf{s}, \mathbf{s}'$ of reachable states of S where $\mathbf{s}'$ is a successor of $\mathbf{s}$ in S.

This representation is analogous to that used in SAT-based model checking (see Chap. 10 of this Handbook [23]) except that the system is formulated in a more powerful logic than propositional logic, though still endowed with efficient satisfiability checkers: SMT solvers. The SMT setting has a number of advantages. To start, the first-order language allows natural and more or less direct formulations of the system under analysis—regardless of whether the system has finitely or infinitely many states. In the finite-state case, these formulations can also be exponentially more compact than propositional ones because they do not need to encode non-Boolean data types and their operations at the propositional level, which allows for better scalability. Moreover, several SAT-based model-checking techniques lift naturally, although not necessarily immediately, to the SMT case.

BMC and k-Induction-Based Methods The most obvious example of such lifting is bounded model checking (BMC) [57]. As in the original propositional setting, one tries to disprove that a given state property $P[\mathbf{x}]$ is *invariant* for the system, i.e., true in all reachable states, by looking for a value $i \geq 0$ such that the formula

$$I[\mathbf{x}_0] \wedge Tr[\mathbf{x}_0, \mathbf{x}_1] \wedge \cdots \wedge Tr[\mathbf{x}_{i-1}, \mathbf{x}_i] \wedge \neg P[\mathbf{x}_i] \tag{1}$$

is satisfiable [4, 93, 125]. Another example is k-induction [148], where one tries to prove that a given state property $P[\mathbf{x}]$ is invariant by looking for a $k \geq 0$ such that (1) is unsatisfiable for all $i = 0, \ldots, k$ and the formula

$$Tr[\mathbf{x}_0, \mathbf{x}_1] \wedge \cdots \wedge Tr[\mathbf{x}_k, \mathbf{x}_{k+1}] \wedge P[\mathbf{x}_0] \wedge \cdots \wedge P[\mathbf{x}_k] \wedge \neg P[\mathbf{x}_{k+1}] \tag{2}$$

is also unsatisfiable. In both examples, the only differences with the original formulations are that the formulas (1) and (2) are first-order qffs; propositional satisfiability is replaced by T-satisfiability; and an SMT solver is used to perform the satisfiability check. Again, as in the propositional case, any variable assignment that satisfies (1) can be used to construct a counter-example trace for P. Several enhancements to BMC and k-induction (such as lemma learning, abstraction and refinement, path compression, termination checks, ...) lift to the SMT case as well [93, 125].

Interpolation-Based Methods Interpolation-based model checking proves a property $P[\mathbf{x}]$ invariant by constructing a formula $R[\mathbf{x}]$ that holds in all reachable states and entails $P[\mathbf{x}]$. This is done incrementally, for $i = 0, 1, \ldots$, by checking the satisfiability of formulas of the form

$$R_i[\mathbf{x}_0] \wedge Tr[\mathbf{x}_0, \mathbf{x}_1] \wedge \cdots \wedge Tr[\mathbf{x}_{k-1}, \mathbf{x}_k] \wedge \big(\neg P[\mathbf{x}_0] \vee \cdots \vee \neg P[\mathbf{x}_k]\big) \tag{3}$$

for some $k > 0$, where R_i is a formula satisfied by all states reachable in up to i steps, starting with $R_0 = I$. When (3) is unsatisfiable, $R_i[\mathbf{x}]$ is generalized to

$R_{i+1} := R_i[\mathbf{x}] \vee Int[\mathbf{x}]$ where $Int[\mathbf{x}_1]$ is a formula entailed by $R_i[\mathbf{x}_0] \wedge Tr[\mathbf{x}_0, \mathbf{x}_1]$ and jointly unsatisfiable with $Tr[\mathbf{x}_1, \mathbf{x}_2] \wedge \cdots \wedge Tr[\mathbf{x}_{k-1}, \mathbf{x}_k] \wedge (\neg P[\mathbf{x}_0] \vee \cdots \vee \neg P[\mathbf{x}_k])$, an *interpolant* of those two formulas. The property is proved if at some point R_{i+1} is equivalent to R_i, something that can be checked by verifying the unsatisfiability of $R_{i+1}[\mathbf{x}] \wedge \neg R_i[\mathbf{x}]$. This method was developed originally in the SAT setting [115]. However, it immediately lifts to the SMT setting with theories T and language fragments for which T-entailment is decidable and interpolants exist and are computable (e.g., [116]). Note that in this case a plain SMT solver is not enough, since procedures for computing theory interpolants are also needed. These procedures, however, can often be built within existing SMT solvers (see Sect. 11.8). See also Chap. 14 for a comprehensive treatment of interpolation-based methods [118].

Predicate-Abstraction-Based Methods Perhaps the most successful approach to software model checking so far, described in more detail in Chap. 15, is predicate abstraction. In a predicate abstraction method popularized by the SLAM model checker and further improved in other tools [8, 95], a program written in a high-level programming language (such as C or Java) and a safety property P to be checked are modeled as a system $(I[\mathbf{x}], Tr[\mathbf{x}, \mathbf{x}'])$ with a distinguished error state directly reachable from any state that violates the property.

The system (I, Tr) is abstracted to a finite-state system $\overline{S} = (\overline{I}, \overline{Tr})$, obtained, roughly speaking, by replacing *predicates* (i.e., atoms or other sub-formulas) of I and Tr by propositional variables. Then, using traditional symbolic model-checking techniques (see Chap. 8 of this Handbook [48]), an exhaustive analysis of all the paths of $\overline{S}$ is performed to determine whether the abstract error state is reachable. If a trace t to that state is found, it is converted to a formula φ that is T-satisfiable exactly when t corresponds to an execution of the original program that leads to the concrete error state. If φ is not T-satisfiable, the abstraction $\overline{S}$ is refined using techniques like those described in Chap. 15 to remove that spurious error trace t. As in the other methods above, all T-satisfiability checks involved in this process are performed by an SMT solver.

11.3 The Lazy Approach to SMT

The majority of the work in SMT has focused on the T-satisfiability of quantifier-free formulas and on theories T for which this problem is decidable. We discuss that major case here. Let us start by observing that to decide the quantifier-free T-satisfiability problem, it is enough in principle to have a procedure for deciding the T-satisfiability of constraints (conjunctions of literals): one can first convert any quantifier-free formula to Disjunctive Normal Form, and then check each disjunct individually. This solution, however, is impractical because of the frequent exponential blow-up in the size of the resulting DNF formula. Except for degenerate (and uninteresting) examples of theories, this blow-up cannot be eliminated in general because the T-satisfiability of qffs is NP-hard, even if the constraint T-satisfiability problem is polynomial, as one can easily show by simple reductions from SAT.

To avoid the inefficiencies inherent in DNF conversions, most current SMT solvers follow a general approach that essentially amounts to constructing and checking a DNF for the input formula incrementally and as needed. The main characteristic of this approach, referred to as the *lazy approach* in the literature (e.g., [147]), is the combination of one or more specialized constraint satisfiability procedures, or *theory solvers*, with a conflict-driven clause-learning (CDCL) SAT solver, the *SAT engine*, used to reason efficiently about the propositional connectives. The approach has several variants, differing in the sophistication of the interaction between the SAT engine and the theory solvers. We discuss some of them in the following.

For the rest of the section, we fix a generic Σ-theory T and assume the existence of a theory solver, or *T-solver* for short, that can decide the T-satisfiability of conjunctions of Σ-literals. We will discuss only a few general desirable features of T-solvers here. Details on algorithms and techniques for implementing theory solvers for specific theories of interest in model checking are provided in Sect. 11.4. We will assume that the reader has some familiarity with the inner workings of modern SAT solvers (see Chap. 9 for a general overview).

11.3.1 A Basic Lazy SMT Solver

In the most basic version of the lazy approach, with a single Σ-theory T, one abstracts each atom in the input formula by a new propositional variable (as detailed at the end of Sect. 11.1.1), uses the SAT engine to find a *model* of the formula, a satisfying assignment given as a set A of literals, and then asks the T-solver to verify that the Σ-literals abstracted by this model are jointly T-satisfiable [19, 124]. If the latter check succeeds, one can conclude that the input formula is T-satisfiable. Otherwise, one asks the SAT engine for another model—something achievable in the simplest way by adding a proper *blocking lemma*, the negation of a subset of the assignment A, to the original formula and restarting the engine. This process is repeated until a model consistent with the theory is found, or all possible propositional models have been explored with no success—in which case one can conclude that the input formula is T-unsatisfiable.

A pseudo-code description of this algorithm is provided in Fig. 1, with the concretization and abstraction functions $(\_)^c$ and $(\_)^a$ defined as in Sect. 11.1.1. Current implementations are based on more sophisticated variations on this basic approach that exploit advanced features of modern SAT engines and theory solvers to achieve a tighter integration between them [3, 5, 31, 72, 77]. The most important ones are described next.

11.3.2 SAT Engine and Theory Solver Features

For efficient integration, in addition to having all the features usually found in modern SAT solvers, it is important for the SAT engine to be *on-line*, i.e., able to take

Require: φ is a qff in the signature Σ of T
Ensure: output is sat if φ is T-satisfiable, and unsat otherwise
  $F := \varphi^{\mathrm{a}}$
  loop
    $A :=$ get_model(F)
    if $A =$ none **then**
      return unsat
    else
      $\mu :=$ check_sat$_T(A^{\mathrm{c}})$
      if $\mu =$ sat **then**
        return sat
      else
        $F := F \wedge \neg\mu^{\mathrm{a}}$

Fig. 1 A basic SMT solver based on the lazy approach. The function get_model implements the SAT engine. It takes a propositional formula F and returns either none, if F is unsatisfiable, or a satisfiable conjunction A of propositional literals such that $A \models F$. The function check_sat$_T$ implements the theory solver. It takes a conjunction ψ of Σ-literals and returns either sat or a T-unsatisfiable conjunction μ of literals from ψ

and process its input progressively, maintaining at all times a set Γ of input formulas and a satisfying assignment for it. Initially, Γ is empty (and so satisfied by the empty assignment). When a new formula is fed as input, the engine attempts to modify the current assignment to satisfy the new formula as well, terminating if that is not possible or waiting for more input formulas otherwise.

T-solvers usually maintain internally at all times a set Λ of literals to be checked for T-satisfiability. The salient advanced features for these solvers are listed below.

Incrementality Intuitively, a T-solver is *incremental* if it can be given a set of literals one at a time and determine each time the T-satisfiability of the newly expanded internal set Λ with a cost proportional to the size of the addition—as opposed to the size of Λ. With an incremental T-solver, the model produced by the SAT engine can be checked for T-satisfiability as it is being constructed, and so discarded as soon as it becomes T-unsatisfiable. Decision procedures used for most theories were either already incremental in their original formulation or have been adapted to be so by SMT researchers.

Backtrackability Incremental solvers are naturally state-based. A state-based T-solver is *backtrackable* if, for any of the literals in its current input set L, it is able to restore inexpensively the state it had right before it was fed that literal. This feature is crucial to keep an incremental T-solver in sync with the SAT engine, which itself relies on backtracking to recover from propositional conflicts generated while attempting to construct a model for its input formula.

Conflict Set Generation A *conflict set* for a T-unsatisfiable input set Λ to a T-solver is an (ideally minimal) subset $\{l_1, \ldots, l_n\}$ of Λ that is already T-unsatisfiable. The T-valid formula $\neg l_1 \vee \cdots \vee \neg l_n$ constructed from this set is called a *justification* or *explanation* (of Λ's unsatisfiability). An explanation can be abstracted and passed to the SAT engine, to be treated as a learned clause. Its immediate effect is to create a conflict in the engine and force a backtrack. If it

is kept afterwards, its later effect will be the same as that of learned lemmas in CDCL SAT solvers: to drive the search away from other parts of the search space that would generate the same conflict.

Literal Deduction A T-solver with this feature is able to identify consequences of its current set Λ among a predetermined set L of literals—i.e., identify literals $l \in L$ such that $\Lambda \models_T l$. This information is useful to the SAT engine which then does not have to guess the value of these literals. Typically, but not exclusively, L consists of all atoms occurring in the original input formula (the formula φ in Fig. 1), as well as their negation. *Theory propagation*, the process of communicating entailed literals to the SAT engine, can be partial or exhaustive, depending on the cost of determining all entailed literals of L. For some theories, such as for instance difference logic (cf. Sect. 11.4.5), exhaustive theory propagation is extremely cheap and proves highly effective. For others, it pays off performance-wise to propagate only literals of L that happened to be deduced in the process of checking the satisfiability of the input set Λ. For instance, this is the case for the (positive) equalities computed by congruence closure in solvers for the theory of equality (cf. Sect. 11.4.1).

Explanation Generation With theory propagation, the SAT engine may generate a conflict involving a theory-propagated literal l. For the engine to perform its conflict analysis and determine how far to backtrack, it is necessary to have an *explanation* for l, a formula of the form $l_1 \wedge \cdots \wedge l_n \rightarrow l$ where $\{l_1, \ldots, l_n\}$ is a subset of the literals Λ in the T-solver such that $l_1, \ldots, l_n \models_T l$. Typically, the same mechanisms and infrastructure used to compute conflict sets can be used to compute these explanations too.[9] Explanations need not be minimal, as computing those can be unacceptably expensive, but should be relatively small since shorter explanations usually lead to better conflict analysis than longer ones. A complication and main difference with conflict sets is that literal explanations are (best) computed *a posteriori* and as needed, whereas conflict sets are usually computed as soon as the input set becomes unsatisfiable (see [132] for a discussion).

11.3.3 A General Framework and Architecture

SMT solvers implementing the many variants of the lazy approach can be described abstractly and declaratively in terms of a transition system between states of the form $M \parallel F$, where M is a sequence of Σ-literals and *decision points*, and F is a quantifier-free Σ-formula in conjunctive normal form, or, equivalently, a set of clauses; an additional distinguished state *Fail* is used to model the discovery by the SMT solver that its input formula is T-unsatisfiable [131, 132]. Identifying for simplicity every Σ-literal l with its propositional abstraction l^{a}, the sequence

[9] In fact, one can look at a conflict set as an explanation for the literal $\bot$.

M represents the propositional assignment being built by the SAT engine, together with the engine's non-deterministic guesses; the formula F models the current set of clauses being processed by the SMT solver. Slightly more concrete versions of this framework also model conflict analysis and lemma construction by adding states of the form $M \parallel F \parallel C$ where M and F are as before and C is a *conflict clause* for M and F, a clause T-entailed by F and propositionally falsified by M [108, 142].

This declarative framework has been used to provide a clean formulation of the different lazy variants and a basis for comparison and formal analysis at an abstraction level free of inessential implementation and control details. Its description, however, is beyond the scope of this chapter. We refer the reader to the original work [108, 132] or previous survey work [12] for more information and formal correctness results. Here, we informally describe instead a general modular architecture for SMT solvers based on the lazy approach, known in the literature as DPLL(T).

The DPLL(T) Architecture The architecture relies on a generic CDCL-style SAT engine, called DPLL(X), which is parametric in the theory and theory solver used. Instantiating the parameter X with a theory solver for some theory T produces a *DPLL(T) system* that can be seen as a concrete implementation of the abstract framework mentioned above.[10] In particular, the engine maintains the partial assignment M and the current formula F. The T-solver maintains a set Λ of literals—which at any time is a subset of those in M. The T-solver can be arbitrary as long as it conforms to a specific, simple interface. The precise details of the interface are not needed here (the interested reader is referred to [77, 132]). It suffices to know that the T-solver provides operations that the DPLL(X) engine can invoke to do the following.

- Assert a literal l, that is, ask for l to be added to Λ. This operation is to be invoked when the DPLL(X) engine adds l to its partial assignment M.
- Ask whether the current set Λ of asserted literals is T-unsatisfiable. This request can be made by the DPLL(X) engine with different degrees of *strength*: for theories where deciding unsatisfiability is expensive, it can be more effective for the engine to rely on a cheap, if incomplete, T-unsatisfiability check while it is building the partial assignment M, and request a complete one only when M propositionally satisfies F (and Λ contains all the literals in M). In response, when it determines the T-unsatisfiability of Λ, the T-solver returns an explanation of that.
- Request a set of input literals not in Λ that are T-entailed by Λ. The returned set, which is used for theory propagation, need not include all T-entailed literals. Note that for this operation the T-solver must know the set of all input literals.
- Request an explanation for a previously theory-propagated literal l. Explanations are used by the DPLL(X) engine during the analysis of conflicts that involve theory-propagated literals.

[10] The motivation for the abbreviation DPLL in DPLL(T) is historical. At the time the architecture was proposed [77], CDCL solvers were still commonly referred to as DPLL solvers—in reference to the work of Davis, Putnam, Logemann and Loveland [62, 63].

- Request the T-solver to *undo* the n most recent assertions, that is, to remove from Λ its n most recent literals, for some $n > 0$. This operation is to be invoked after the engine backtracks to some previous decision level and shrinks its partial assignment M correspondingly.

DPLL(T) is currently the most popular general architecture for SMT solvers based on the lazy approach. However, its black-box treatment of the SAT engine and the theory solvers (originally an asset because it allowed the use of minimally modified off-the-shelf SAT solvers) is becoming a limitation as research in SMT advances. A number of alternative architectures have been proposed quite recently that aim at overcoming these limitations by integrating propositional-level and theory-level reasoning more tightly [34, 98, 119].

11.4 Theory Solvers for Specific Theories

In this section, we consider solvers for constraint satisfiability problems in several specific theories. For each theory, we first describe the signature and semantics of the theory and then discuss how to solve its constraint satisfiability problem.

11.4.1 Uninterpreted Function Symbols

We start with the simplest possible theory consisting of a given signature Σ and the class of all Σ-models. This theory, or rather family of theories parametrized by the signature, is known as the theory of *Equality with Uninterpreted Functions* (*EUF*) or the *empty theory*—since it imposes no restrictions on its models.

Conjunctions of literals in this theory can be decided in polynomial time by congruence closure algorithms. For simplicity, we describe a version of the algorithm assuming no predicate symbols. This assumption loses no generality, because predicate symbols can be handled using a simple encoding: introduce a new sort symbol B and a new function symbol f_p of rank $\sigma_1 \cdots \sigma_n \mathsf{B}$ for each predicate symbol p of rank $\sigma_1 \cdots \sigma_n$, plus a new constant symbol tt of sort B; then, replace each literal $p(t_1, \ldots, t_n)$ with $f_p(t_1, \ldots, t_n) = \mathsf{tt}$ and each literal $\neg p(t_1, \ldots, t_n)$ with $f_p(t_1, \ldots, t_n) \neq \mathsf{tt}$.

Let Φ be a set of literals to be checked for satisfiability. Since there are no predicate symbols, Φ can be partitioned into a set E of equalities and a set D of disequalities. Let E^* be the *congruence closure* of E, defined as the smallest equivalence relation (over the terms in Φ) that includes E and also satisfies the congruence property: for every pair of terms $f(s_1, \ldots, s_n)$ and $f(t_1, \ldots, t_n)$, $(f(s_1, \ldots, s_n), f(t_1, \ldots, t_n)) \in E^*$ whenever $(s_i, t_i) \in E^*$ for $i = 1, \ldots n$.[11] Then, Φ is satisfiable iff for each $t_1 \neq t_2 \in D$, $(t_1, t_2) \notin E^*$.

[11] Observe that two terms may be related by E^* only if they have the same sort.

Example 1 Let $\Phi = \{f(f(a)) = a,\ f(f(f(a))) = a,\ g(a) \neq g(f(a))\}$. The equivalence classes induced by E are $\{a,\ f(f(a)),\ f(f(f(a)))\}$, $\{f(a)\}$, $\{g(a)\}$, $\{g(f(a))\}$. Congruence closure requires merging the first two classes and the last two. As a result, $(g(a),\ g(f(a))) \in E^*$ and Φ is not satisfiable.

Standard algorithms use directed acyclic graphs (DAGs) to represent terms, and a *union-find* data structure [155] to represent equivalence classes of terms. The main work is in the congruence closure step. A simple $O(n^2)$ algorithm for congruence closure is as follows [127]: seed a work-list with the equalities in E; then, while the work-list is non-empty, remove an equality, perform a *union* operation on the two equivalence classes containing the terms on either side of the equality, and then examine all pairs of parents (in the DAG) of these terms to see if any of them newly satisfy the congruence property; if they do, add this new pair to the work-list. Once the work-list is empty, Φ is satisfiable iff for each disequality $t_1 \neq t_2 \in D$, the *find* of t_1 is different from the *find* of t_2. More efficient algorithms ($O(n \log n)$) only require traversing one set of parents after each *union* operation and include efficient mechanisms for computing, in the case when Φ is unsatisfiable, a small unsatisfiable subset of Φ [66, 130].

11.4.2 Real Arithmetic

Next, consider the signature Σ containing a single sort, R, all rational number constants, function symbols $\{+, -, *\}$ and the predicate symbol $\leq$, all with the expected rank. The theory of real arithmetic consists of this signature paired with the standard model of the real numbers, that is the Σ-model that interprets R as the set $\mathbb{R}$ of the real numbers and the constants and operators in the usual way. Satisfiability of Σ-formulas in this theory, even with quantifiers, is decidable [112]. Traditionally, decision procedures for the full theory have not been efficient enough to be practical. It is worth noting, however, that this is an area of active research and several promising new approaches are being investigated [80, 99]. Efficient decision procedures do exist for the satisfiability of appropriately restricted classes of quantifier-free Σ-formulas in this theory.

Consider, for instance, *linear real arithmetic* (LRA). Here, formulas are restricted in that the symbol $*$ can only appear if at least one of its two operands is a rational constant. For illustration purposes, we describe here a simple algorithm based on Fourier–Motzkin elimination [146]. For convenience, let $t_1 < t_2$ abbreviate $\neg(t_2 \leq t_1)$ and assume that in all constraints, like terms are combined.

Now, suppose we are given a set Φ of LRA literals. We first eliminate disequalities by replacing $t_1 \neq t_2$ by $t_1 < t_2 \vee t_2 < t_1$. We also eliminate weak inequalities by replacing $t_1 \leq t_2$ with $t_1 < t_2 \vee t_1 = t_2$. These steps introduce disjunctions, but case-splitting or conversion to DNF can be used to reduce the new problem to several instances of simple conjunctions of strict inequalities. Next, we eliminate equalities. If $t_1 = t_2$ cannot be solved for some variable x, it must either be trivially true or

trivially false. If the former, we remove it; if the latter, Φ is unsatisfiable and we are done. Otherwise, the equality is equivalent to $x = t_3$ for some term t_3. In this case, we replace x everywhere by t_3 and then remove the equality.

We are left with only conjunctions of strict inequalities, to which we apply Fourier–Motzkin elimination. We pick a variable x occurring in Φ to eliminate, and rewrite all constraints containing x as either (i) $t_1 < x$ or (ii) $x < t_2$. For every possible pair of constraints in Φ consisting of a constraint of the form (i) and one of the form (ii), we introduce the new constraint $t_1 < t_2$. We then remove all constraints containing x, eliminating x from Φ. We repeat with another variable until no more variables appear in Φ. The result is a set of inequalities over rational constants that can easily be simplified to $\bot$, indicating that Φ is unsatisfiable, or $\neg\bot$, indicating that Φ is satisfiable.

Example 2 Let $\Phi = \Phi_1 \cup \{w \leq y\}$ with $\Phi_1 = \{x < y + z,\ x - y = z - w,\ y < 0\}$. Eliminating $\leq$ yields two sets of constraints: $\Phi_1 \cup \{w < y\}$ and $\Phi_1 \cup \{w = y\}$. In the first set, solve the equality for x to get $x = y + z - w$. After substituting and combining like terms, we have $\{0 < w,\ y < 0,\ w < y\}$. Applying Fourier–Motzkin elimination to y results in $\{0 < w,\ w < 0\}$. Then eliminating w yields $0 < 0$, which simplifies to $\bot$. For the second set of constraints, first eliminate $w = y$ by substituting y for w everywhere to get $\{x < y + z,\ x = z,\ y < 0\}$. Next, eliminate x which gives $\{0 < y,\ y < 0\}$. Fourier–Motzkin elimination on y then again yields $0 < 0$. Thus Φ is unsatisfiable.

Each elimination step in the procedure above introduces in the worst case a quadratic number of new constraints, making the procedure doubly exponential. For this reason, Fourier–Motzkin elimination is usually not practical for large sets of constraints, Though more efficient procedures based on Fourier–Motzkin have been developed [98, 106], procedures based on the Simplex method are currently preferred because of their superior overall performance. A Simplex-based algorithm specialized for use in SMT solvers is given in [67], and further improvements on it are described in [68, 91, 104].

11.4.3 Integer Arithmetic

Consider now a signature Σ containing a single sort Z, for the integers, all integer number constants, function symbols $\{+, -, *\}$ and the predicate symbol $\leq$, all with the expected rank. The theory of integer arithmetic consists of this signature paired with the standard model of the integers, the Σ-model that interprets Z as the set $\mathbb{Z}$ of the integers, and the constants and operators in the usual way. The satisfiability of Σ-formulas in this theory, even without quantifiers, is undecidable [112].

The *linear integer arithmetic* (LIA) fragment is the analog of the LRA fragment described above: the symbol $*$ can only appear if at least one of its operands

is an integer constant. The fully quantified LIA fragment is also known as *Presburger arithmetic* and is decidable using Presburger's algorithm [138]. More efficient methods exist for the quantifier-free fragment. Again, for illustrative purposes, we describe here a relatively simple procedure for quantifier-free LIA based on the Omega test [21, 107, 140]. This is essentially an integer adaptation of the Fourier–Motzkin elimination procedure described for the reals above.

Let Φ be a set of LIA literals. As before, we assume that all constraints are normalized by combining like terms. We divide coefficients in each constraint by any common factors and check for any contradictions in constraints involving only constants. Then, we eliminate disequalities by replacing $t_1 \neq t_2$ with $t_1 < t_2 \vee t_2 < t_1$, where again $s < t$ abbreviates $\neg(t \leq s)$. We similarly eliminate weak inequalities by replacing $(t_1 \leq t_2)$ with $t_1 < t_2 + 1$.

The next step is the elimination of equalities. If it is possible to solve some equation for some variable x, we do this and either: (i) halt the procedure and report Φ is unsatisfiable if the right-hand side of the solved equation reduces to a non-integer constant; or else (ii) substitute the right-hand side for x in Φ as before. If it is not possible to solve any equation for a variable while maintaining integer coefficients, we proceed as follows: let a be the minimum coefficient of any variable and write the equation it appears in as $ax + \sum_i a_i x_i + c = 0$. Let $m = |a| + 1$ and define $k \mathbin{\widehat{mod}} m = k - m\lfloor \frac{k}{m} + \frac{1}{2} \rfloor$. Note that $\widehat{mod}$ distributes (modulo m) over both multiplication and addition. Next, apply this operator to both sides of the original equation to get: $\pm(x \mathbin{\widehat{mod}} m) + \sum_i (a_i \mathbin{\widehat{mod}} m)(x_i \mathbin{\widehat{mod}} m) + (c \mathbin{\widehat{mod}} m) = 0$ (modulo m). Expanding the definition of $\widehat{mod}$, this can be rewritten as: $\pm x + \sum_i (a_i \mathbin{\widehat{mod}} m) x_i + (c \mathbin{\widehat{mod}} m) = m \cdot y$ where y is a fresh variable. This equation can now be used to eliminate x from the original equation (and indeed from Φ). The new equation still has the same number of variables, since y was introduced, but in the new equation, the absolute values of the coefficients of all variables other than y are reduced by a factor of at least $2/3$, while the absolute value of the coefficient of y is in fact $|a|$. By repeating this process a logarithmic number of times, we eventually obtain an equation in which some variable has coefficient ± 1 and can thus be eliminated without introducing any new variables. This process can be used to eliminate all equality constraints from Φ.

The final step again involves only conjunctions of strict inequalities and is similar to Fourier–Motzkin elimination. We pick a variable x occurring in Φ to eliminate, and write all constraints containing x as either (i) $ax < t_1$ or (ii) $t_2 < bx$ where a and b are positive integers. We remove these constraints from Φ and then for every possible pair consisting of a constraint of the form (i) and a constraint of the form (ii), we add a new constraint, choosing from the following three alternatives: the *real shadow* $at_2 < bt_1$; the *dark shadow* $bt_1 - at_2 > ab$; and the *gray shadow* $\bigvee_{i=1}^{i=b-1} bx = t_2 + i$. The first two are approximations, with the first preserving the soundness and the second the completeness of the procedure. The gray shadow is exact and can be used to eliminate x via additional case splitting and equation solving. However, this can be prohibitively expensive, so one possible strategy is: check whether the real shadow constraints are sufficient for unsatisfiability; failing that, check whether the dark shadow constraints are satisfiable; and finally, failing that, check the gray shadow constraints.

Example 3 Let $\Phi = \{2x = 3w + 4z,\ w < x,\ 2x + 4z < w\}$. When solving for x, the minimal coefficient is 2, so $m = 3$, and we can derive the new equation $-x - z = 3y$, or $x = -3y - z$. Substituting into the first equation, we get $-6y - 2z = 3w + 4z$, or $w = -2y - 2z$. Substituting for x and w in the second constraint, we get $-2y - 2z < -3y - z$ or $y < z$. Substituting for x and w in the third constraint, we get $2(-3y - z) + 4z < -2y - 2z$ or $z < y$. The real shadow is now unsatisfiable.

As with real arithmetic, better performance is possible by using Simplex-based algorithms, in this case expanded with additional techniques for obtaining integer solutions [65, 67, 68, 91, 92].

11.4.4 Mixed Integer and Real Arithmetic

Sometimes it is desirable to mix integer and real reasoning. A simple solution is to use two different sorts, Z and R and then create two copies of the arithmetic symbols, one set for integers and one set for reals. Mapping operators, such as toInt of rank RZ (returning the integer part of a real) and toReal of rank ZR (returning the corresponding real) can be used to mix real and integer terms.

Alternatively, mixing can be done by reasoning within the theory of reals with the addition of a unary predicate symbol Int whose interpretation is exactly the set of all whole (real) numbers. Often, constraints of interest limit the use of the Int predicate to variables (as opposed to more complicated terms). In such cases, an algorithm can be obtained by mixing approaches for LRA and LIA [21, 68].

For example, suppose Φ is a set of literals. Let V_{Z} be the set of variables in Φ that are constrained by Int, and let V_{R} be the set of remaining variables of Φ. We can eliminate disequalities and weak inequalities as before. Then, all equations that contain at least one variable in V_{R} can be eliminated by solving for the variable and then substituting for that variable in Φ. Next, the remaining variables in V_{R} can be eliminated by performing Fourier–Motzkin elimination. The result of this step is a system of equalities and inequalities over only the variables in V_{Z}. Furthermore, each constraint can be made to have integer coefficients by multiplying through by the least common multiple of the denominators appearing in its rational coefficients. The resulting set of constraints can be solved using any algorithm for LIA, such as the one described above.

11.4.5 Difference Logic

Difference logic refers to a quantifier-free arithmetic fragment in which all atoms are of the form $x - y \bowtie c$, where $\bowtie \in \{=, \leq, \geq\}$, c is a constant, and x and y are variables. The background theory may be the theory of real arithmetic, in which case c can be any rational constant and the fragment is called *real difference logic*

(RDL). Alternatively, it may be the theory of integer arithmetic, in which case c is required to be an integer constant and the fragment is called *integer difference logic*. Conjunctions of literals in either IDL or RDL can be solved in polynomial time. A simple algorithm is as follows [128].

Let $t_1 < t_2$ abbreviate $\neg(t_2 \leq t_1)$, and eliminate disequalities by replacing $\neg(x - y = c)$ with $x - y < c \vee x - y > c$. We similarly eliminate equalities by replacing $(x - y = c)$ with $x - y \leq c \wedge x - y \geq c$. Finally, we write all constraints in terms of $\leq$ by applying the following rewriting steps: (i) $x - y \geq c \longrightarrow y - x \leq -c$; (ii) $x - y > c \longrightarrow y - x < -c$; and (iii) $x - y < c \longrightarrow x - y \leq c - 1$. Step (iii) is only valid in IDL. For RDL, a slight variation is possible: $x - y < c \longrightarrow x - y \leq c - \delta$ where δ is a rational positive constant chosen to be sufficiently small [146].

Now, we form a weighted directed graph with a vertex for each variable and an edge from x to y with weight c for each constraint $x - y \leq c$. The set of constraints is satisfiable iff there is no cycle for which the sum of the weights on the edges is negative, which can be determined using standard graph algorithms [50].

Example 4 Let $\Phi = \{x - y = 5, z - y \geq 2, z - x > 2, w - x = 2, z - w < 0\}$. After eliminating equality and rewriting, we have $\{x - y \leq 5, y - x \leq -5, y - z \leq -2, x - z \leq -3, w - x \leq 2, x - w \leq -2, z - w \leq -1\}$. In the associated graph, the cycle from x to z to w to x has total weight -2. Therefore, Φ is unsatisfiable.

The algorithm described here is elaborated and extended in [58, 161]. An efficient alternative algorithm based on a reduction to propositional logic is described in [103].

11.4.6 Bit Vectors

The theory of *fixed-size bit vectors* is useful for modeling hardware or low-level software. The theory signature consists of one sort BV_n for each bit width $n \geq 1$; 2^n binary constants for each such sort, each representing a constant bit vector of width n; and a large set of operators corresponding to standard hardware and software operations on bit vectors. For example, $t_1 \circ t_2$ represents the *concatenation* of bit vectors t_1 and t_2 and $t[i : j]$ represents the *extraction* of bits i through j of t, where $n > i \geq j \geq 0$ and n is the bit width of t.

A conjunction of equations containing only concatenation and extraction operators can be checked for satisfiability in polynomial time as follows [41, 61]. In step (i), simple rewrites are used to distribute extraction over concatenation, other extractions, or constants, until the only arguments of extractions are variables. In step (ii), whenever an equation contains a concatenation on one side, $r \circ s = t$, it is replaced by two equations: $r = t[n - 1 : m]$ and $s = t[m - 1 : 0]$, where n is the bit width of t and m is the bit width of s. In step (iii), if a variable x appears as an argument to two different extractions, $x[i : j]$ and $x[k : l]$, with $i > k \geq j$, then $x[i : j]$ is replaced by $x[i : k + 1] \circ x[k : j]$. Similarly, if $i > l \geq j$, then $x[i : j]$ is replaced

by $x[i:l+1] \circ x[l:j]$. These three steps are repeated until they can no longer be applied. Let $\sim$ be the equivalence relation over terms induced by the resulting set of equations. The original equations are unsatisfiable iff there exist two distinct binary constants, c_1 and c_2, such that $c_1 \sim c_2$.

Example 5 Let x be of width 4 and consider the equation $1 \circ x = x \circ 0$. Step (ii) produces three new equations: $1 = x[3:3]$, $x[0:0] = 0$, and $x[3:1] = x[2:0]$. Then, step (iii) requires that the last equation be replaced with $x[3:3] \circ x[2:1] = x[2:1] \circ x[0:0]$. Repeating step (ii) on this equation gives $x[3:3] = x[2:2]$, $x[2:2] = x[1:1]$, and $x[1:1] = x[0:0]$. The equivalence relation induced by all of these equations equates 0 and 1, so the original equation is unsatisfiable.

Almost any extensions beyond this core fragment of the theory, including just allowing disequalities, make the constraint satisfiability problem NP-hard. Recent results show that, depending on the extension, the problem can be NP-complete, PSPACE-complete, or up to NEXPTIME-complete for the full fragment [74]. Solvers typically handle the general case by first employing a set of rewrite rules to simplify and normalize parts of the input and then encoding the result as a propositional satisfiability problem. This can be done by assigning a propositional variable to each bit in each bit vector variable and then using propositional logic formulas to encode each equation in terms of these variables—a process known as *bit blasting*. In reality, the situation is more nuanced as several bit blasting SMT solvers, including non-DPLL(T) solvers such as Boolector and STP, bit blast some of their internal formulas only as needed, and so combine aspects of both the lazy and the eager approaches.

Both the rewrite rules and the method of encoding can dramatically affect performance, as detailed in an extensive set of publications on the subject [7, 18, 25, 36, 37, 44, 76, 94, 113].

11.4.7 Arrays

Consider a signature Σ with sorts $\mathsf{A}, \mathsf{I}, \mathsf{E}$ (for *arrays*, *indices* and array *elements*) and function symbols: read, of rank $\mathsf{A}\,\mathsf{I}\,\mathsf{E}$ and write of rank $\mathsf{A}\,\mathsf{I}\,\mathsf{E}\,\mathsf{A}$. Then, consider the theory consisting of all Σ-structures satisfying the axioms:

1. $\forall a{:}\mathsf{A}\,\forall i{:}\mathsf{I}\,\forall v{:}\mathsf{E}\ \mathsf{read}(\mathsf{write}(a,i,v),i) = v$,
2. $\forall a{:}\mathsf{A}\,\forall i,j{:}\mathsf{I}\,\forall v{:}\mathsf{E}\ i \neq j \Rightarrow \mathsf{read}(\mathsf{write}(a,i,v),j) = \mathsf{read}(a,j)$,
3. $\forall a\,\forall b{:}\mathsf{A}\ (\forall i{:}\mathsf{I}\,\mathsf{read}(a,i) = \mathsf{read}(b,i)) \Rightarrow a = b$.

This is the theory of functional arrays with extensionality. (Axiom (3) may be omitted to obtain a theory without extensionality.) This theory is especially useful for modeling memories or array data structures. The full theory is undecidable although it contains a number of decidable fragments [33].

A simple algorithm for constraint satisfiability can be obtained by naive instantiation of the axioms plus the use of congruence closure (e.g., [101]). Let Φ be a set of

Σ-literals. With no loss of generality, assume that each element of Φ is a *flat literal*, that is, of the form $a = b$, $a \neq b$, $v = \mathsf{read}(a, i)$, or $b = \mathsf{write}(a, i, v)$, where a, b, i, v are variables.[12] First, replace any disequality $a \neq b$ between array variables with $\mathsf{read}(a, k) \neq \mathsf{read}(b, k)$, where k is a fresh variable of sort I. Now, let I be the set of all variables in Φ of sort I, and replace each formula of the form $a = \mathsf{write}(b, i, v)$ with $\mathsf{read}(a, i) = v \wedge \bigwedge_{j \in I}(i = j \vee \mathsf{read}(a, j) = \mathsf{read}(b, j))$. Since this step introduces disjunctions, case-splitting or conversion to Disjunctive Normal Form (DNF) can be used to reduce the new problem to several instances of sets of literals. Each such instance can be checked for satisfiability using only congruence closure over read, since write no longer appears in Φ. The set Φ is satisfiable iff one of these instances is satisfiable.

Example 6 Let $\Phi = \{\mathsf{read}(a, i) = v, \mathsf{read}(b, i) \neq v, w \neq v, a = \mathsf{write}(b, j, w)\}$. The reduction replaces the last equation with $\mathsf{read}(a, j) = w \wedge (i = j \vee \mathsf{read}(a, i) = \mathsf{read}(b, i))$. Now, note that if $i = j$, then congruence closure generates $\mathsf{read}(a, i) = \mathsf{read}(a, j)$ and so $v = w$, contradicting $w \neq v$. On the other hand, if $\mathsf{read}(a, i) = \mathsf{read}(b, i)$, then this contradicts $\mathsf{read}(b, i) \neq v$. Thus, Φ is unsatisfiable.

In practice, various heuristics and optimizations are used to avoid many unnecessary axiom instantiations, greatly reducing the number of cases that must be considered [27, 35, 76, 89, 122, 153].

11.4.8 Other Theories

There are many other theories of general interest with decision procedures that have been or could be integrated into SMT solvers. These include theories of finite sets [47], finite multi-sets [136], inductive data types [13] (lists, records, and tuples can be handled as special cases), character strings [102], pointers [49, 111], and floating point numbers [144]. It is also possible to design special-purpose theories for specific application domains (see, e.g. [129]).

11.5 Combining Theory Solvers

All constraint satisfiability procedures described in the previous section consider theories of a single data type. In many applications of SMT, however, including model checking, one is often interested in the satisfiability of formulas over several data types (e.g., arrays with integer indices and real values, lists of integers, etc.)

[12] Any set of literals can be converted into an equisatisfiable set of flat literals by introducing new variables.

and, consequently, over some combination of their theories. An important theoretical and practical question then is whether and how constraint satisfiability procedures for different theories can be combined modularly into a single one so as to allow the construction of theory solvers for a combination of these theories. This section gives an overview of notable combination methods and results.

A general mechanism for combination is available when the desired combination of theories is axiomatized simply by the union of the axioms of the individual theories.[13] More formally, since theories here are defined as classes of models, a modular combination of two theories (the combination of more theories is similar) is defined as follows.

Let $T_1 = (\Sigma_1, \mathbf{A}_1)$ and $T_2 = (\Sigma_2, \mathbf{A}_2)$ be two theories such that Σ_1 and Σ_2 agree on the rank they assign to their shared function and predicate symbols.[14] The *combination* of T_1 and T_2 is the theory $T_1 \oplus T_2 = (\Sigma_1 \oplus \Sigma_2, \mathbf{A})$ where $\Sigma_1 \oplus \Sigma_2$ is the smallest supersignature of both Σ_1 and Σ_2, and $\mathbf{A} = \{\mathscr{A} \mid \mathscr{A}^{\Sigma_1,\emptyset} \in \mathbf{A_1}$ and $\mathscr{A}^{\Sigma_2,\emptyset} \in \mathbf{A_2}\}$. These definitions encompass the more traditional view of theories defined by a set of axioms. In particular, if $\mathbf{A}_i$ is the class of all Σ_i models that satisfy a set $\mathbf{Ax}_i$ of Σ_i-sentences for $i = 1, 2$, then $\mathbf{A}$ above is the class of all $\Sigma_1 \oplus \Sigma_2$-models that satisfy the set $\mathbf{Ax}_1 \cup \mathbf{Ax}_2$ [141, 158]. Given, for $i = 1, 2$, a decision procedure for the satisfiability of sets of Σ_i-literals in a Σ_i-theory T_i, we are interested in constructing a decision procedure for the satisfiability of sets of $\Sigma_1 \oplus \Sigma_2$-literals in $T_1 \oplus T_2$ using those procedures as black boxes.

11.5.1 A Basic Combination Method

A combination method originally due to Nelson and Oppen [126], and later adapted and extended to sorted logics by others [87, 141, 159], provides a general mechanism for combining decision procedures as above. Variants of the method are implemented in all major SMT solvers. Its essence is captured by the following nondeterministic procedure.

The Nelson–Oppen Procedure Let Γ be a set of literals in the combined signature $\Sigma_1 \oplus \Sigma_2$. (i) First, *purify* Γ by constructing an equisatisfiable literal set $\Gamma_1 \cup \Gamma_2$ where each Γ_i consists of Σ_i-literals only. This can be easily done by finding a pure (i.e., Σ_i- for some i) subterm t, replacing it with a new variable v, adding the equation $v = t$ to the set, and then repeating this process until all literals are pure. (ii) Then get the component satisfiability procedures to agree on the values assigned to the *shared variables*[15] of Γ_1 and Γ_2, the variables appearing in both Γ_1 and Γ_2.

[13] An example of a theory which is *not* a modular combination in this sense is the theory of finite sets with cardinality. This theory includes the theory of finite sets and the theory of integers, but also additional, *mixed* axioms defining the cardinality operator.

[14] Shared symbols with (same name but) different ranks can always be renamed apart.

[15] Also called *interface variables* in recent literature—see, e.g., [38].

This is done by guessing an *arrangement of* V, that is, a set $arr(V)$ of equations and disequations encoding an equivalence relation over V (and so expressing which pairs of variables take the same value and which do not). (iii) Finally, check each Γ_i locally for T_i-satisfiability under the chosen arrangement.

If each satisfiability procedure finds its respective input $\Gamma_i \cup arr(V)$ satisfiable, report the original set Γ to be $T_1 \oplus T_2$-satisfiable. Otherwise, repeat steps (ii) and (iii) with another arrangement. If no suitable arrangement exists, report Γ to be $T_1 \oplus T_2$-unsatisfiable. □

The non-deterministic combination procedure above yields a decision procedure for a large class of theories. Its main requirement is that T_1 and T_2 be *disjoint* in the sense of not sharing any function or predicate symbols. The procedure is terminating simply because the purification step is terminating and the number of possible arrangements is finite (although exponential in the number of shared variables). The procedure is *refutationally sound* for any two disjoint theories: every set it declares $T_1 \oplus T_2$-unsatisfiable is indeed so. Without additional restrictions the method is not *refutationally complete*, as it may fail to detect the unsatisfiability of its input for certain pairs of disjoint theories [160]. It becomes complete if both T_1 and T_2 are *stably infinite* over the sorts they share [134, 157, 159]. A Σ-theory T is stably infinite over a sort σ in Σ if every T-satisfiable quantifier-free Σ-formula is satisfiable in a T-interpretation that interprets σ as an infinite set.

Many theories of interest in SMT applications are indeed stably infinite over some or all of their sorts. Examples include the various theories of arithmetic discussed in Sect. 11.4 and the theory of arrays, which is stably infinite over its array, index and element sorts. However, some are not—most notably the theory of bit vectors described in Sect. 11.4.6.

With disjoint stably infinite theories the combination method has an exponential worst-case time complexity in general. More precisely, if the constraint satisfiability problem for T_i can be decided in time $O(f_i(n))$ for $i = 1, 2$, the corresponding problem for $T_1 \oplus T_2$ can be decided in time $O(2^{n^2} \times (f_1(n) + f_2(n)))$, with the exponential factor due to the need to guess the right arrangement [134].

11.5.2 Combination Variants and Extensions

Actual implementations of the non-deterministic procedure sketched above try to reduce its exponential penalty by reducing the amount of guessing of arrangements. The most common approach is to check the satisfiability of Γ_1 and Γ_2 locally and then *deduce and propagate*, from one component decision procedure to the other, disjunctions of shared equalities entailed by Γ_1 or Γ_2. This is particularly effective when T_1 and T_2 are both *convex* over the sorts they share because then it is enough for completeness to consider only individual entailed equalities. A Σ-theory T is convex over a sort $\sigma \in \sigma^S$ if for all sets Φ of Σ-literals and all sets E of equalities between variables of sort σ, $\Phi \models_T \bigvee_{e \in E} e$ iff $\Phi \models_T e$ for some $e \in E$.

With convex theories, worst-case time complexity of the combination goes down to $O(n^4 \times (f_1(n) + f_2(n)))$ [134].

With non-convex theories, or convex theories for which computing entailed equalities is expensive, another approach is to check the T_i-satisfiability of Γ_i alone for some $i = 1, 2$ and, once a model $\mathscr{A}_i$ is found, make the optimistic assumption that $\Gamma_j \cup arr(V)$ is T_j-satisfiable, where $j \neq i$ and $arr(V)$ is the arrangement of V induced by $\mathscr{A}_i$. If $\Gamma_j \cup arr(V)$ is unsatisfiable because of some of the literals in $arr(V)$, a new model for Γ_i with different truth values for those literals must be found. For some theory combinations, this heuristic approach is highly effective in practice [123].

The stable infiniteness requirement can be relaxed for one theory if the other satisfies stronger properties [109, 141, 160]. However, the equality-sharing mechanism of the original combination procedure needs to be extended to certain cardinality constraints. The most general results so far in the context of many-sorted logic are described in [97]. A case for using a typed logic with parametric types to frame and generalize Nelson–Oppen combination is provided in [109]. A few extensions have been proposed to lift the disjointness restriction, most notably by Ghilardi and his collaborators, although their interest thus far has been mostly theoretical [84, 87, 158]. Recent work, however, uses Ghilardi's results to develop novel SMT-based LTL model-checking algorithms [85, 86, 88].

11.6 SMT Solving Extensions and Enhancements

The scope of SMT solvers, especially those based on the lazy approach, has been extended further in a number of directions. Also, several solvers contain further enhancements aimed at improving their overall performance. We briefly describe a few significant extensions and enhancements next.

Multiple Theories When working with multiple theories $T_1, \ldots, T_n$ that can be combined with the Nelson–Oppen method, one can generate a single theory solver for their combination T by combining the constraint satisfiability procedures for various theories, as described in Sect. 11.5. With solvers based on the DPLL(T) architecture a better approach is to develop an independent theory solver for each theory and extend the interface of the SAT engine so that it interacts directly with each theory solver and coordinates among them in Nelson–Oppen style, to maintain soundness and completeness.

A general framework for doing this is known as *delayed theory combination* (*DTC*) [30, 38]. At the level of abstract transition systems described in Sect. 11.3.3, the essence of DTC is to work again with states of the form $M \parallel F$ except that now every atom occurring in M or in F is *pure*, i.e., in the signature of one of the theories $T_1, \ldots, T_n$. A preprocessing purification step can be applied to the SMT solver's input to guarantee this for the initial formula F_0. The atoms in M come from F_0 or from the set S of all *interface equalities*, equalities between variables

that occur in two atoms of F_0 belonging to different theories. The SAT engine is modified so that it also determines, by guessing, the truth values of the atoms in S, in addition to those in F_0. In its more general and advanced form, DTC also benefits from changes to the theory solvers that enable them to propagate entailed interface equalities or disjunctions of them, thus reducing the SAT engine's guesswork. More details on DTC together with a study of its relative merits with respect to the encapsulation approach mentioned at the beginning can be found in [38]. A general abstract formulation of multi-theory lazy SMT that encompasses DTC is provided in [108].

Quantifiers Checking the satisfiability of quantified formulas is a long-standing challenge in SMT. A typical use of quantifiers in input formulas is to provide axiomatic definitions for function or predicate symbols not in the solver's background theories. In model-checking applications, other uses of quantifiers include assertions involving all the elements of a collection datatype (such as arrays and sets) and assertions about concurrent systems (which for instance quantify over all processes). Extending decision procedures to such quantified formulas without losing termination is in general impossible because of basic undecidability results for first-order logic. In fact, even maintaining (refutational) completeness is already difficult, both in theory and in practice.

While the T-satisfiability of quantified formulas is decidable for certain theories T (such as, for instance, the theory of real numbers), their decision procedures use *quantifier elimination* methods, which convert formulas into T-equivalent quantifier-free ones, and are quite heavy computationally. Furthermore, these methods normally break down in the presence of additional symbols, such as uninterpreted ones. As a consequence, SMT solvers use incomplete methods based on instantiating quantified formulas into a set of *ground* ones.[16] Existential quantifiers in formulas of the form $\forall x_1 \cdots \forall x_n \, \exists x \, \varphi$ (with $n \geq 0$) are eliminated by dropping $\exists x$ and replacing all free occurrences of x in φ by the term $f(x_1, \ldots, x_n)$ where f is a fresh (uninterpreted) function symbol of arity n. Then, each universally quantified formula $\forall x \, \varphi$ is conjoined with a number of its *instances*, obtained from the qff φ by replacing its free occurrences of x with some ground term of the same sort. The selection of these instances is driven by incomplete heuristics.

The most common strategy is to select for instantiation ground terms that are *relevant* to $\forall x \, \varphi$, according to some heuristic relevance criterion. The SMT solver tries to find a subterm $t[x]$ of $\forall x \, \varphi$ properly containing x, a ground term g among those in its working memory, and a subterm s of g, such that $t[s]$, the result of replacing x by s in t, is T-equivalent to g. The expectation is that instantiating x with s is more likely to be helpful than instantiating it with an arbitrary ground term. In terms of unification theory [6], checking that $\models_T t[s] = g$ is a special case of *T-matching*. In the context of SMT, because of the richness of the background theory T, it may be very difficult if not impossible to determine whether an arbitrary term T and a

[16] A term or formula is ground if it contains no variables (although it may contain uninterpreted constant symbols).

ground term g T-match. As a result, most implementations use some form of T-matching only for uninterpreted terms. More details on this and on heuristics that are fairly effective in practice can be found in [24, 64, 82].

More recent work has focused on identifying fragments of first-order logic modulo theories for which is it possible to produce complete, and in some cases also terminating, quantifier-instantiation methods [70, 83, 150]. Some of this work [83] is based on a general *model-based quantifier instantiation* approach where the SMT solver maintains at all times (a finite representation of) a *candidate model*, a T-model that satisfies the current set G of ground formulas. The solver uses the candidate model to focus instance generation on only a few ground instances falsified by that model. Unless extending G with these instances makes G unsatisfiable, the solver then constructs a new candidate model for the extended G, and repeats the process until it is able to construct one that satisfies all quantified formulas as well.

A similar idea is used in the Inst-Gen calculus for first-order logic (with no theories) [78]. New ground instances are generated based on a model for the ground ones computed by an off-the-shelf SAT solver. A theorem prover based on this calculus has been shown to be very effective [105]. The Inst-Gen calculus has been extended to built-in theories [79]. However, implementing the extended calculus in an efficient solver has proven difficult so far.

A recent and quite promising line of work on model-based instantiation focuses on SMT formulas all of whose quantifiers range over uninterpreted sorts [142, 143]. There, the solver tries to prove its input formula T-satisfiable by imposing finite cardinality constraints on those sorts, identifying for each sort σ a set U_σ of ground terms that enumerates the sort's finite domain, and instantiating quantifiers with these terms. The candidate model is used also to avoid exhaustive instantiation over each U_σ by identifying, and ignoring, whole sets of instances that are equisatisfiable with an already generated one.

Layered Theory Solvers Some theories T with a decidable constraint satisfiability problem contain less-expressive fragments whose constraint satisfiability problem can be decided by more efficient methods. For example, the theory of real numbers includes a chain of increasingly larger and harder to decide fragments: inequalities between variables, difference constraints, linear constraints, and nonlinear constraints. For these theories, one can design a *layered* T-solver consisting of a sequence of subsolvers of decreasing performance but increasing generality [5, 32, 37, 59, 149]. In principle then, the solver can use the most efficient subsolver that is able to process the conjunction of literals given as input.

In reality, inputs rarely fall neatly in one of the fragments in the sequence. So abstraction and refinement techniques, similar in spirit to those used in model checking, must be used to take advantage of the faster subsolvers. Considering a nonincremental theory solver, for simplicity, the layering mechanism works as follows. The solver abstracts the literals in the input formula ψ as needed to get a formula ψ' T-entailed by ψ and accepted by its most restricted subsolver. If that subsolver

determines ψ' to be T-unsatisfiable, the solver reports ψ to be T-unsatisfiable.[17] Otherwise, it refines ψ' just enough for it to fall into the fragment processed by the next more general subsolver, and sends it to that one, repeating the same process until a subsolver finds the refined formula ψ' unsatisfiable or ψ' gets refined to ψ.

Incomplete Theory Solvers For some theories—such as the theories of arrays, linear integer arithmetic, algebraic data types, and finite sets—the constraint satisfiability problem alone is NP-hard. To be refutationally complete then, a solver for such a theory T must perform internal search and case splitting. In a DPLL(T) setting, it is possible to use much simpler, albeit incomplete, T-solvers by delegating all search and case splitting to the SAT engine, a module already designed to do that efficiently.

The main idea, developed in the *splitting on demand* framework [11], is the following. Any time the T-solver needs to do a case analysis to determine the T-satisfiability of its input, it encodes the needed case split into a T-valid disjunction of literals, a theory lemma in effect. Then, instead of returning a sat/unsat answer, it returns the lemma demanding that the SAT engine process it—doing case splits on it as it would do with any other lemma. When the engine adds a literal from the lemma to its variable assignment, the literal will be asserted back to the T-solver, letting it proceed with that choice. After that, the T-solver either manages to determine the satisfiability of the newly extended input set or repeats the process with a new lemma. For termination, and overall completeness, there must be a finite upper bound on the number of splitting demands a T-solver needs to make for any given input before it is able to reply with sat or unsat. General sufficient conditions for the correctness of splitting on demand are discussed in [11].

Although splitting on demand simplifies the construction of theory solvers, it does not always provide the best performance. A discussion of this issue for real arithmetic solvers can be found in [92].

11.7 Eager Encodings to SAT

An alternative to the lazy approach to SMT is one usually referred to as the *eager* approach. It encompasses any technique that aims to fully reduce SMT problems to propositional satisfiability (SAT) problems via some kind of encoding. More formally, an eager SMT solver accepts a first-order formula φ in the signature of some theory T, generates a propositional formula ψ that is propositionally satisfiable iff φ is T-satisfiable, and then it feeds ψ to an off-the-shelf SAT solver.

Although the *lazy* approach is now predominant in SMT, mostly because of its flexibility and generality, efficient eager solvers do exist for a number of important theories. To give a sense of how some of them work, let us look at the theory of Equality with Uninterpreted Functions (EUF), that was introduced in Sect. 11.4.1.

[17] If a conflict set is required for ψ, it can be computed from one for ψ'.

Eager solvers for quantifier-free formulas in EUF can be constructed using *Ackermann's reduction* [1]. Suppose f is a unary function symbol (the generalization to n-ary symbols is straightforward) in an input formula φ, and let $\{f(t_1), \ldots, f(t_n)\}$ be the set of occurrences of f in φ. We introduce n new constant symbols $f_1, \ldots, f_n$ and replace each $f(t_i)$ with f_i in φ. Let φ' denote the result of this replacement. Then the formula $\varphi' \wedge \bigwedge_{i=1}^{n} \bigwedge_{j=i+1}^{n} (t_i = t_j \Rightarrow f_i = f_j)$ is satisfiable in EUF iff φ is. By repeating this process, all function symbols can be removed.[18]

To complete the eager translation, we must also remove all equality literals. One way to do this is by introducing propositional variables and transitivity constraints [46]. Suppose ψ is an EUF formula with equalities but no function symbols. Let $S = \{s_1, \ldots, s_m\}$ be the set of all terms appearing in equalities. Let ψ' be the result of replacing each equality $s_i = s_j$ or $s_j = s_i$ where $i < j$ with a propositional variable $e_{i,j}$. Let G be an undirected graph on S with an edge between s_i and s_j iff $e_{i,j}$ appears in ψ'. Let G' be a *chordal* graph (no chord-free cycle of size four or more) obtained from G by adding arbitrary chords within cycles of size four or more. For each *triangle* (s_i, s_j, s_k), i.e., cycle of size three in G' with $i < j < k$, we add the following *transitivity constraint* to ψ': $((e_{i,j} \wedge e_{j,k}) \Rightarrow e_{i,k}) \wedge ((e_{i,j} \wedge e_{i,k}) \Rightarrow e_{j,k}) \wedge ((e_{i,k} \wedge e_{j,k}) \Rightarrow e_{i,j})$. The result is a propositional formula that is satisfiable iff ψ is satisfiable in EUF. For additional details on and extensions to this algorithm, see [107].

Example 7 Consider again the EUF example $\Phi = \{f(f(a)) = a,\ f(f(f(a))) = a,\ g(a) \neq g(f(a))\}$. After applying the Ackermann reduction, we have: $\{f_2 = a,\ f_3 = a,\ g_4 \neq g_5,\ a = f_1 \Rightarrow f_1 = f_2,\ a = f_2 \Rightarrow f_1 = f_3,\ f_1 = f_2 \Rightarrow f_2 = f_3,\ a = f_1 \Rightarrow g_4 = g_5\}$. The graph G is already chordal and has four triangles: (a, f_1, f_2), (a, f_1, f_3), (a, f_2, f_3), (f_1, f_2, f_3). Let $a_0 \equiv a$ and introduce the propositional terms $e_{i,j}$ according to the subscripts. Also, let $B_{i,j,k}$ be the transitivity constraint on $e_{i,j}$, $e_{j,k}$, $e_{i,k}$ introduced above. The set $\{e_{0,2},\ e_{0,3},\ \neg e_{4,5},\ e_{0,1} \Rightarrow e_{1,2},\ e_{0,2} \Rightarrow e_{1,3},\ e_{1,2} \Rightarrow e_{2,3},\ e_{0,1} \Rightarrow e_{4,5},\ B_{0,1,2},\ B_{0,1,3},\ B_{0,2,3},\ B_{1,2,3}\}$ of propositional formulas is satisfiable iff Φ is satisfiable. It is easy to see that this set is unsatisfiable: $e_{0,2}$ and $e_{0,3}$ must be true and $e_{4,5}$ must be false. The fifth constraint then implies that $e_{1,3}$ must also be true. But then $B_{0,1,3}$ entails $e_{0,1}$ which implies that $e_{4,5}$ must be true, a contradiction.

The UCLID solver [42, 45, 110] uses these and other techniques to solve (eagerly) problems specified in the *CLU logic*, a logic of Counter arithmetic with Lambda expressions and Uninterpreted functions. Other eager approaches have looked at small-domain instantiations [137] and various fragments of arithmetic [151, 152]. As mentioned in Sect. 11.4.6, a common approach to construct solvers for the theory of bit vectors is to apply some rewriting to the input formula followed by bit blasting. This too is an instance of the eager approach.

[18] A method due to Bryant can be used as an alternative that can sometimes be more efficient [43].

11.8 Additional Functionalities of SMT Solvers

Arguably, the success of SMT solvers as embedded deductive reasoning engines is due in large part to the emergence of additional functionalities well beyond the mere checking of a formula's T-satisfiability. These functionalities are used extensively and with great benefit by tools such as model checkers, interactive provers, program verifiers, test case generators and so on. We discuss a selection of them next.

Models In many applications it is useful not only to know that a formula is T-satisfiable but also to obtain a witness of its T-satisfiability in the form of a T-interpretation (in the sense of Sect. 11.1.1) satisfying the formula. Fully representing first-order models such as T-interpretations finitarily, however, is challenging, when possible at all. Hence SMT solvers usually return only partial information, in the form of value assignments to selected symbols in the input formula. Furthermore, they restrict consideration only to models that permit a finitary representation of these values.[19] For instance, for the theory of arrays they only consider models that interpret array variables as *almost constant maps*, unary functions mapping all their inputs to the same value except for finitely many inputs. A similar restriction is adopted for EUF in computing the interpretation of function symbols.

Even with these restrictions, returning models may require strictly more work than just determining satisfiability. Depending on the theory, different approaches are possible. One approach, followed for instance by the CVC3 solver [17], is first to compute a partial model sufficient to establish the input formula's satisfiability, and then to do additional work as needed to extend that partial model to include values for symbols of interest (variables and function/predicate symbols) to the user. Another approach, followed for instance by the solvers Yices and Z3, is to instrument the theory solver to maintain some value for every symbol at all times, starting with some default assignment, and modifying the assignment as needed until it becomes a satisfying one. Yet another approach, which is implemented in CVC4 and is beneficial with quantified formulas whose quantifiers range only over uninterpreted sorts, is to explicitly construct models that interpret those sorts as finite sets [142].

Proofs For most applications that utilize SMT solvers, it is important to have confidence in their refutational soundness. Since proving the soundness of an SMT solver is unrealistic, due to the complexity of such tools, a reasonable approach is for the solver to accompany its unsat answers with a *certificate* that can be checked independently with much simpler and more trustworthy tools. This certificate is in general a proof of the input formula's unsatisfiability, expressed as a proof term in some suitable proof system.

With SMT solvers based on the lazy approach it is possible to produce proofs with a two-tiered structure, consisting of a propositional skeleton filled with several theory-specific subproofs. In these two-tiered proofs, the conclusion is reached by

[19] These restrictions cause no loss of generality with quantifier-free queries.

means of propositional inferences applied to a set of input formulas and *theory lemmas*. The latter are disjunctions of theory literals deduced from no assumptions, using proof rules specific to the background theory in question. The propositional skeleton is generated using techniques similar to those used by proof-producing SAT solvers (e.g., [2, 9]). The proofs of the various lemmas used as hypotheses in the propositional skeleton are produced typically using natural deduction inference rules with theory-specific axioms [28, 73, 81, 120].

A major challenge for the field is to devise a common proof system for proof-producing SMT solvers. The wide diversity of theories and solving algorithms in SMT makes it difficult to find a single proof system that is universally good. One way to address this difficulty is to use a meta-language for specifying proof systems for SMT [154]. The advantage of a meta-language solution is that one can build an automatic proof checker generator that takes as input a proof system and generates an efficient proof checker for that system [133].

Unsatisfiable Cores Most SMT solvers allow the user to inquire about the joint T-satisfiability of a set of formulas. For T-unsatisfiable sets Φ, some solvers are able to return a *T-unsatisfiable core*, a possibly minimal subset of Φ that is also T-unsatisfiable. This functionality, which is useful in many applications, is patterned after the analogous one offered at the propositional level by many modern SAT solvers. Research on producing minimal or small T-unsatisfiable cores in SMT is not as extensive as in SAT. Current methods either are inspired by similar ones in the SAT literature or rely directly on propositional technology. Following Barrett et al.'s terminology [12], we can identify three main approaches.

In the *proof-based approach*, adopted by proof-producing SMT solvers such as CVC3 and MathSAT, a T-unsatisfiable core is extracted from the produced proof of unsatisfiability simply by collecting all the formulas of Φ that appear as premises in the proof. The size of the returned core depends on the sophistication of the proof-generation mechanism in producing compact proofs. This approach requires only a small additional implementation effort but incurs the (heavy) cost of producing a proof, even when none is requested.

In the *assumption-based approach*, implemented in Yices and applicable to any DPLL(T)-style solver, the input set $\Phi = \{\varphi_1, \ldots, \varphi_n\}$ is internally converted into the equisatisfiable set $\{p_1 \Rightarrow \varphi_1, \ldots, p_n \Rightarrow \varphi_n, p_1, \ldots, p_n\}$ where each p_i is a fresh propositional symbol. Then, the same conflict analysis mechanism used by the DPLL engine can be used to identify a subset of $\{p_1, \ldots, p_n\}$ that caused the last conflict. The returned T-unsatisfiable core consists of the corresponding φ_i's.

In the *lemma-lifting approach* [56], implemented in more recent versions of MathSAT, one uses the fact that a DPLL(T) solver will discover the T-unsatisfiability of Φ by adding theory lemmas until Φ becomes propositionally unsatisfiable. Once the DPLL engine detects unsatisfiability, any external propositional core extractor can be used to produce an unsatisfiable core C for the propositional abstraction $\{\varphi^{\mathrm{a}} \mid \varphi \in \Phi\}$ of the extended Φ. The returned T-unsatisfiable core consists then of $\{\varphi \mid \varphi^{\mathrm{a}} \in C\}$, the formulas of Φ whose abstraction is in C.

Interpolants A fundamental result in model theory due to Craig [60] asserts the existence of an *interpolant* for every pair of first-order formulas A and B such that $A \models B$. This is a formula I written using only logical symbols and symbols occurring in both A and B such that $A \models I$ and $I \models B$. Analogues of this result, expressed in terms of unsatisfiability instead of entailment, hold for a variety of logics and logic fragments. In the SMT case, the result states that for all first-order theories T and formulas A, B such that $A, B \models_T \bot$, there is a formula I using only logical symbols, symbols of T and symbols occurring in both A and B such that $A \models_T I$ and $I, B \models_T \bot$.

Starting with the seminal work by McMillan [115], interpolants have found a number of practical uses in model checking (see Chap. 14). Applications involve the computation of interpolants in propositional logic or in logics with (combinations of) theories such as the theory of equality, linear rational arithmetic, arrays, and finite sets [54, 100, 117, 162].

In propositional logic, interpolants can be computed from resolution proofs using a simple method due to Pudlák [139]. For theories T with the *quantifier-free interpolation property*, which guarantees the existence of quantifier-free interpolants for any T-unsatisfiable pair A, B of quantifier-free formulas, interpolants can be computed using SMT techniques. In many cases, it is possible to produce interpolants efficiently by modifying existing theory solvers in relatively minor ways [54, 75, 145].

Under fairly general conditions, the generation of theory interpolants for sets of literals can be extended modularly to (i) sets of arbitrary quantifier-free formulas and (ii) combinations of theories (each with the quantifier-free interpolation property), thanks to a method by Yorsh and Musuvathi [162]. This allows one to turn an SMT solver into an interpolant generator. The first extension is possible with SMT solvers that produce the sort of two-tiered proofs mentioned earlier in this section, and relies on an adaptation of Pudlák's method to deal with the proof's propositional skeleton. The second extension additionally requires each component theory T to be *equality-interpolating*: whenever $A, B \models_T r = t$ where r is a term occurring in A and t a term occurring in B, it is possible to compute a term s in the language shared by A and B such that $A, B \models_T r = s \wedge s = t$. A further, and related, requirement is that the unsatisfiability proof from which the interpolant is extracted contains no AB*-mixed* literals, literals with symbols occurring only in A and symbols occurring only in B. Unfortunately, typical SMT solvers do not guarantee the absence of AB-mixed literals from their proofs.[20] Initial implementations of the Yorsh–Musuvathi method imposed restrictions on solver search strategies in order to produce proofs of a certain shape from which it is possible to extract interpolants even in the presence of AB-mixed literals [54]. In later work, these restrictions, and their potential performance penalty, have been increasingly and considerably reduced by relying on a certain amount of proof post-processing [40, 55, 90] or by considering only certain classes of theories [52].

[20] Both Delayed Theory Combination and Splitting on Demand generate new literals during a proof which may be AB-mixed.

Acknowledgement Clark Barrett was at the Courant Institute of Mathematical Sciences at New York University, New York, NY, USA, when he coauthored this chapter.

References

1. Ackermann, W.: Solvable Cases of the Decision Problem. North-Holland, Amsterdam (1954)
2. Amjad, H.: A compressing translation from propositional resolution to natural deduction. In: Konev, B., Wolter, F. (eds.) Intl. Symp. on Frontiers of Combining Systems (FroCoS). LNCS, vol. 4720, pp. 88–102. Springer, Heidelberg (2007)
3. Armando, A., Castellini, C., Giunchiglia, E.: SAT-based procedures for temporal reasoning. In: Biundo, S., Fox, M. (eds.) European Conf. on Planning (ECP). LNCS, vol. 1809, pp. 97–108. Springer, Heidelberg (2000)
4. Armando, A., Mantovani, J., Platania, L.: Bounded model checking of software using SMT solvers instead of SAT solvers. In: Valmari, A. (ed.) Intl. Workshop on Model Checking Software (SPIN). LNCS, vol. 3925, pp. 146–162. Springer, Heidelberg (2006)
5. Audemard, G., Bertoli, P., Cimatti, A., Korniłowicz, A., Sebastiani, R.: A SAT-based approach for solving formulas over Boolean and linear mathematical propositions. In: Voronkov, A. (ed.) Intl. Conf. on Automated Deduction (CADE). LNCS, vol. 2392, pp. 195–210. Springer, Heidelberg (2002)
6. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press, Cambridge (1998)
7. Babić, D.: Exploiting structure for scalable software verification. Ph.D. thesis, University of British Columbia (2008)
8. Ball, T., Majumdar, R., Millstein, T.D., Rajamani, S.K.: Automatic predicate abstraction of C programs. In: Burke, M., Soffa, M.L. (eds.) Conf. on Programming Language Design and Implementation (PLDI), pp. 203–213. ACM, New York (2001)
9. Barrett, C., Berezin, S.: A proof-producing Boolean search engine. In: Intl. Workshop on Pragmatics of Decision Procedures in Automated Reasoning (PDPAR) (2003)
10. Barrett, C., Conway, C., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011)
11. Barrett, C., Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Splitting on demand in SAT modulo theories. In: Hermann, M., Voronkov, A. (eds.) Intl. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR). LNCS, vol. 4246, pp. 512–526. Springer, Heidelberg (2006)
12. Barrett, C., Sebastiani, R., Seshia, S., Tinelli, C.: Satisfiability modulo theories. In: Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, vol. 185, pp. 825–885. IOS Press, Amsterdam (2009)
13. Barrett, C., Shikanian, I., Tinelli, C.: An abstract decision procedure for a theory of inductive data types. J. Satisf. Boolean Model. Comput. **3**, 21–46 (2007)
14. Barrett, C., Stump, A., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB) (2010). www.smtlib.org
15. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: Version 2.0. In: Intl. Workshop on Satisfiability Modulo Theories (SMT) (2010)
16. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: Version 2.0. Tech. rep., Department of Computer Science, University of Iowa (2010)
17. Barrett, C., Tinelli, C.: CVC3. In: Damm, W., Hermanns, H. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 4590, pp. 298–302. Springer, Heidelberg (2007)
18. Barrett, C.W., Dill, D.L., Levitt, J.R.: A decision procedure for bit-vector arithmetic. In: Chawla, B.R., Bryant, R.E., Rabaey, J.M. (eds.) Design Automation Conf. (DAC), pp. 522–527. ACM, New York (1998)

19. Barrett, C.W., Dill, D.L., Stump, A.: Checking satisfiability of first-order formulas by incremental translation to SAT. In: Brinksma, E., Larsen, K.G. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 2404, pp. 236–249 (2002)
20. Barrett, C.W., de Moura, L., Stump, A.: Design and results of the first satisfiability modulo theories competition (SMT-COMP 2005). J. Autom. Reason. **35**(4), 373–390 (2005)
21. Berezin, S., Ganesh, V., Dill, D.L.: An online proof-producing decision procedure for mixed-integer linear arithmetic. In: Garavel, H., Hatcliff, J. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 2619, pp. 521–536. Springer, Heidelberg (2003)
22. Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press, Amsterdam (2009)
23. Biere, A., Kroening, D.: SAT-based model checking. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
24. Bjørner, N., de Moura, L.: Efficient E-matching for SMT solvers. In: Pfenning, F. (ed.) Intl. Conf. on Automated Deduction (CADE). LNCS, vol. 4603, pp. 183–198. Springer, Heidelberg (2007)
25. Bjørner, N., Pichora, M.C.: Deciding fixed and non-fixed size bit-vectors. In: Steffen, B. (ed.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 1384, pp. 376–392. Springer, Heidelberg (1998)
26. Bobot, F., Conchon, S., Contejean, E., Lescuyer, S.: Implementing polymorphism in SMT solvers. In: Joint Workshops of the Intl. Workshop on Satisfiability Modulo Theories and the Intl. Workshop on Bit-Precise Reasoning (SMT/BPR), pp. 1–5. ACM, New York (2008)
27. Bofill, M., Nieuwenhuis, R., Oliveras, A., Carbonell, E.R., Rubio, A.: A write-based solver for SAT modulo the theory of arrays. In: Cimatti, A., Jones, R.B. (eds.) Formal Methods in Computer Aided Design (FMCAD), pp. 1–8. IEEE, Piscataway (2008)
28. Böhme, S., Weber, T.: Fast LCF-style proof reconstruction for Z3. In: Kaufmann, M., Paulson, L.C. (eds.) Intl. Conf. on Interactive Theorem Proving (ITP). LNCS, vol. 6172, pp. 179–194. Springer, Heidelberg (2010)
29. Bouton, T., De Oliveira, D.C.B., Déharbe, D., Fontaine, P.: veriT: an open, trustable and efficient SMT-solver. In: Schmidt, R.A. (ed.) Intl. Conf. on Automated Deduction (CADE). LNCS, vol. 5663, pp. 151–156. Springer, Heidelberg (2009)
30. Bozzano, M., Bruttomesso, R., Cimatti, A., Junttila, T., Ranise, S., Sebastiani, R., van Rossum, P.: Efficient satisfiability modulo theories via delayed theory combination. In: Etessami, K., Rajamani, S.K. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 3576. Springer, Heidelberg (2005)
31. Bozzano, M., Bruttomesso, R., Cimatti, A., Junttila, T., van Rossum, P., Ranise, S., Sebastiani, R.: Efficient theory combination via Boolean search. Inf. Comput. **204**(10), 1411–1596 (2006)
32. Bozzano, M., Bruttomesso, R., Cimatti, A., Junttila, T., van Rossum, P., Schulz, S., Sebastiani, R.: An incremental and layered procedure for the satisfiability of linear arithmetic logic. In: Halbwachs, N., Zuck, L.D. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 3440, pp. 317–333 (2005)
33. Bradley, A., Manna, Z., Sipma, H.: What's decidable about arrays? In: Emerson, E.A., Namjoshi, K.S. (eds.) Intl. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI). LNCS, vol. 3855, pp. 427–442. Springer, Heidelberg (2006)
34. Brain, M., D'Silva, V., Haller, L., Griggio, A., Kroening, D.: An abstract interpretation of DPLL(T). In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) Intl. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI). LNCS, vol. 7737, pp. 455–475. Springer, Heidelberg (2013)
35. Brummayer, R., Biere, A.: Lemmas on demand for the extensional theory of arrays. In: Joint Workshops of the Intl. Workshop on Satisfiability Modulo Theories and the Intl. Workshop on Bit-Precise Reasoning (SMT/BPR), pp. 6–11. ACM, New York (2008)

36. Brummayer, R., Biere, A.: Boolector: an efficient SMT solver for bit-vectors and arrays. In: Kowalewski, S., Philippou, A. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 5505, pp. 174–177. Springer, Heidelberg (2009)
37. Bruttomesso, R., Cimatti, A., Franzén, A., Griggio, A., Hanna, Z., Nadel, A., Palti, A., Sebastiani, R.: A lazy and layered SMT(BV) solver for hard industrial verification problems. In: Damm, W., Hermanns, H. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 4590, pp. 547–560. Springer, Heidelberg (2007)
38. Bruttomesso, R., Cimatti, A., Franzén, A., Griggio, A., Sebastiani, R.: Delayed theory combination vs. Nelson-Oppen for satisfiability modulo theories: a comparative analysis. Ann. Math. Artif. Intell. **55**(1–2), 63–99 (2009)
39. Bruttomesso, R., Pek, E., Sharygina, N., Tsitovich, A.: The openSMT solver. In: Esparza, J., Majumdar, R. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 6015, pp. 150–153. Springer, Heidelberg (2010)
40. Bruttomesso, R., Rollini, S., Sharygina, N., Tsitovich, A.: Flexible interpolation with local proof transformations. In: Scheffer, L., Phillips, J.R., Hu, A.J. (eds.) Intl. Conf. on Computer-Aided Design, pp. 770–777. IEEE, Piscataway (2010)
41. Bruttomesso, R., Sharygina, N.: A scalable decision procedure for fixed-width bit-vectors. In: Roychowdhury, J.S. (ed.) Intl. Conf. on Computer-Aided Design, pp. 13–20. ACM, New York (2009)
42. Bryant, R., Lahiri, S., Seshia, S.: Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In: Brinksma, E., Larsen, K.G. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 2404, pp. 106–122. Springer, Heidelberg (2002)
43. Bryant, R.E., German, S., Velev, M.N.: Exploiting positive equality in a logic of equality with uninterpreted functions. In: Halbwachs, N., Peled, D. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 1633, pp. 470–482. Springer, Heidelberg (1999)
44. Bryant, R.E., Kroening, D., Ouaknine, J., Seshia, S.A., Strichman, O., Brady, B.: Deciding bit-vector arithmetic with abstraction. In: Grumberg, O., Huth, M. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 4424, pp. 358–372. Springer, Heidelberg (2007)
45. Bryant, R.E., Lahiri, S.K., Seshia, S.A.: Deciding CLU logic formulas via Boolean and pseudo-Boolean encodings. In: Intl. Workshop on Constraints in Formal Verification (CFV) (2002)
46. Bryant, R.E., Velev, M.N.: Boolean satisfiability with transitivity constraints. ACM Trans. Comput. Log. **3**(4), 604–627 (2002)
47. Cantone, D., Zarba, C.: A new fast tableau-based decision procedure for an unquantified fragment of set theory. In: Caferra, R., Salzer, G. (eds.) Automated Deduction in Classical and Non-classical Logics. LNCS, vol. 1761, pp. 492–495. Springer, Heidelberg (2000)
48. Chaki, S., Gurfinkel, A.: BDD-based symbolic model checking. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
49. Chatterjee, S., Lahiri, S., Qadeer, S., Rakamarić, Z.: A reachability predicate for analyzing low-level software. In: Grumberg, O., Huth, M. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 4424, pp. 19–33. Springer, Heidelberg (2007)
50. Cherkassy, B.V., Goldberg, A.V.: Negative-cycle detection algorithms. In: Díaz, J., Serna, M.J. (eds.) Annual European Symp. on Algorithms (ESA). LNCS, vol. 1136, pp. 349–363. Springer, Heidelberg (1996)
51. Christ, J., Hoenicke, J., Nutz, A.: SMTinterpol: an interpolating SMT solver. In: Donaldson, A.F., Parker, D. (eds.) Intl. Workshop on Model Checking Software (SPIN). LNCS, vol. 7385, pp. 248–254. Springer, Heidelberg (2012)
52. Christ, J., Hoenicke, J., Nutz, A.: Proof tree preserving interpolation. In: Piterman, N., Smolka, S. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 7795, pp. 124–138. Springer, Heidelberg (2013)

53. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT solver. In: Piterman, N., Smolka, S.A. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 7795, pp. 93–107. Springer, Heidelberg (2013)
54. Cimatti, A., Griggio, A., Sebastiani, R.: Efficient interpolant generation in satisfiability modulo theories. In: Ramakrishnan, C.R., Rehof, J. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 4963, pp. 397–412. Springer, Berlin (2008)
55. Cimatti, A., Griggio, A., Sebastiani, R.: Efficient generation of Craig interpolants in satisfiability modulo theories. ACM Trans. Comput. Log. **12**(1), 1–54 (2010)
56. Cimatti, A., Griggio, A., Sebastiani, R.: Computing small unsatisfiable cores in satisfiability modulo theories. J. Artif. Intell. Res. **40**(1), 701–728 (2011)
57. Clarke, E.M., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Form. Methods Syst. Des. **19**(1), 7–34 (2001)
58. Cotton, S., Maler, O.: Fast and flexible difference constraint propagation for DPLL(T). In: Biere, A., Gomes, C.P. (eds.) Theory and Applications of Satisfiability Testing (SAT). LNCS, vol. 4121, pp. 170–183. Springer, Heidelberg (2006)
59. Cotton, S., Maler, O.: Satisfiability modulo theory chains with DPLL(T). Tech. rep., Verimag (2006)
60. Craig, W.: Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. J. Symb. Log. **22**(3), 269–285 (1957)
61. Cyrluk, D., Möller, O., Rueß, H.: An efficient decision procedure for the theory of fixed-sized bit-vectors. In: Grumberg, O. (ed.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 1254, pp. 60–71. Springer, Heidelberg (1997)
62. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem proving. Commun. ACM **5**(7), 394–397 (1962)
63. Davis, M., Putnam, H.: A computing procedure for quantification theory. J. ACM **7**(3), 201–215 (1960)
64. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM **52**(3), 365–473 (2005)
65. Dillig, I., Dillig, T., Aiken, A.: Cuts from proofs: a complete and practical technique for solving linear inequalities over integers. In: Maler, A.B.O. (ed.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 5643, pp. 233–247. Springer, Heidelberg (2009)
66. Downey, P.J., Sethi, R., Tarjan, R.E.: Variations on the common subexpression problem. J. ACM **27**(4), 758–771 (1980)
67. Dutertre, B., de Moura, L.: A fast linear-arithmetic solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006)
68. Dutertre, B., de Moura, L.: Integrating simplex with DPLL(T). Tech. rep., CSL, SRI International (2006)
69. Dutertre, B., de Moura, L.: The YICES SMT solver. Tech. rep., SRI International (2006)
70. Echenim, M., Peltier, N.: An instantiation scheme for satisfiability modulo theories. Journal of Automated Reasoning, 293–362 (2010)
71. Enderton, H.B.: A Mathematical Introduction to Logic, 2nd edn. Academic Press, Cambridge (2001)
72. Flanagan, C., Joshi, R., Ou, X., Saxe, J.B.: Theorem proving using lazy proof explication. In: Hunt, W.A. Jr., Somenzi, F. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 2725, pp. 355–367. Springer, Heidelberg (2003)
73. Fontaine, P., Marion, J.Y., Merz, S., Nieto, L.P., Tiu, A.: Expressiveness + automation + soundness: towards combining SMT solvers and interactive proof assistants. In: Hermanns, H., Palsberg, J. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 3920, pp. 167–181. Springer, Berlin (2006)

74. Fröhlich, A., Kovásznai, G., Biere, A.: More on the complexity of quantifier-free fixed-size bit-vector logics with binary encoding. In: Bulatov, A.A., Shur, A.M. (eds.) Computer Science: Theory and Applications. LNCS, vol. 7913, pp. 378–390. Springer, Heidelberg (2013)
75. Fuchs, A., Goel, A., Grundy, J., Krstić, S., Tinelli, C.: Ground interpolation for the theory of equality. In: Kowalewski, S., Philippou, A. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 5505, pp. 413–427. Springer, Heidelberg (2009)
76. Ganesh, V., Dill, D.L.: A decision procedure for bit-vectors and arrays. In: Damm, W., Hermanns, H. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 4590, pp. 519–531. Springer, Heidelberg (2007)
77. Ganzinger, H., Hagen, G., Nieuwenhuis, R., Oliveras, A., Tinelli, C.D.: Fast decision procedures. In: Alur, R., Peled, D. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 3114, pp. 175–188. Springer, Heidelberg (2004)
78. Ganzinger, H., Korovin, K.: New directions in instantiation-based theorem proving. In: Symp. on Logic in Computer Science, vol. LICS, pp. 55–64. IEEE, Piscataway (2003)
79. Ganzinger, H., Korovin, K.: Theory instantiation. In: Hermann, M., Voronkov, A. (eds.) Intl. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR). LNCS, vol. 4246, pp. 497–511. Springer, Heidelberg (2006)
80. Gao, S., Kong, S., Clarke, E.M.: dReal: an SMT solver for nonlinear theories over the reals. In: Bonacina, M.P. (ed.) Intl. Conf. on Automated Deduction (CADE). LNCS, vol. 7898, pp. 208–214. Springer, Berlin (2013)
81. Ge, Y., Barrett, C.: Proof translation and SMT-LIB benchmark certification: a preliminary report. In: Intl. Workshop on Satisfiability Modulo Theories (SMT) (2008)
82. Ge, Y., Barrett, C., Tinelli, C.: Solving quantified verification conditions using satisfiability modulo theories. In: Pfenning, F. (ed.) Intl. Conf. on Automated Deduction (CADE). LNCS, vol. 4603, pp. 167–182. Springer, Heidelberg (2007)
83. Ge, Y., de Moura, L.: Complete instantiation for quantified formulas in satisfiability modulo theories. In: Bouajjani, A., Maler, O. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 5643, pp. 306–320. Springer, Heidelberg (2009)
84. Ghilardi, S.: Model-theoretic methods in combined constraint satisfiability. J. Autom. Reason. **33**(3–4), 221–249 (2005)
85. Ghilardi, S., Nicolini, E., Ranise, S., Zucchelli, D.: Combination methods for satisfiability and model-checking of infinite-state systems. In: Pfenning, F. (ed.) Intl. Conf. on Automated Deduction (CADE). LNCS, vol. 4603, pp. 362–378. Springer, Heidelberg (2007)
86. Ghilardi, S., Nicolini, E., Ranise, S., Zucchelli, D.: Towards SMT model checking of array-based systems. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) Intl. Joint Conf. on Automated Reasoning (IJCAR). LNCS, vol. 5195, pp. 67–82. Springer, Heidelberg (2008)
87. Ghilardi, S., Nicolini, E., Zucchelli, D.: A comprehensive combination framework. ACM Trans. Comput. Log. **9**(2), 1–54 (2008)
88. Ghilardi, S., Ranise, S.: MCMT: a model checker modulo theories. In: Giesl, J., Hähnle, R. (eds.) Intl. Joint Conf. on Automated Reasoning (IJCAR). LNCS, vol. 6173, pp. 22–29. Springer, Heidelberg (2010)
89. Goel, A., Krstić, S., Fuchs, A.: Deciding array formulas with frugal axiom instantiation. In: Joint Workshops of the Intl. Workshop on Satisfiability Modulo Theories and the Intl. Workshop on Bit-Precise Reasoning (SMT/BPR), pp. 12–17. ACM, New York (2008)
90. Goel, A., Krstić, S., Tinelli, C.: Ground interpolation for combined theories. In: Schmidt, R.A. (ed.) Intl. Conf. on Automated Deduction (CADE). LNCS, vol. 5663, pp. 183–198. Springer, Heidelberg (2009)
91. Griggio, A.: An effective SMT engine for formal verification. Ph.D. thesis, DISI, University of Trento (2009)
92. Griggio, A.: A practical approach to SMT(LA(Z)). In: Intl. Workshop on Satisfiability Modulo Theories (SMT) (2010)

93. Hagen, G., Tinelli, C.: Scaling up the formal verification of Lustre programs with SMT-based techniques. In: Cimatti, A., Jones, R.B. (eds.) Formal Methods in Computer Aided Design (FMCAD), pp. 109–117. IEEE, Piscataway (2008)
94. Jha, S.K., Limaye, R.S., Seshia, S.A.: Beaver: engineering an efficient SMT solver for bit-vector arithmetic. Tech. rep., EECS Department, University of California, Berkeley
95. Jhala, R., McMillan, K.L.: Interpolant-based transition relation approximation. In: Etessami, K., Rajamani, S.K. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 3576, pp. 39–51. Springer, Heidelberg (2005)
96. Jhala, R., Podelski, A., Rybalchenko, A.: Predicate abstraction for program verification. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
97. Jovanović, D., Barrett, C.: Polite theories revisited. In: Fermüller, C.G., Voronkov, A. (eds.) Intl. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR). LNCS, vol. 6397, pp. 402–416. Springer, Heidelberg (2010)
98. Jovanović, D., Barrett, C., de Moura, L.: The design and implementation of the model constructing satisfiability calculus. In: Formal Methods in Computer Aided Design (FMCAD), pp. 173–180. IEEE, Piscataway (2013)
99. Jovanović, D., de Moura, L.: Solving non-linear arithmetic. In: Gramlich, B., Miller, D., Sattler, U. (eds.) Intl. Joint Conf. on Automated Reasoning (IJCAR). LNCS, vol. 7364, pp. 339–354. Springer, Heidelberg (2012)
100. Kapur, D., Majumdar, R., Zarba, C.: Interpolation for data structures. In: Young, M., Devanbu, P.T. (eds.) Intl. Symp. on Foundations of Software Engineering (FSE), pp. 105–116. ACM, New York (2006)
101. Kapur, D., Zarba, C.G.: A reduction approach to decision procedures. Tech. rep., University of New Mexico (2005)
102. Kiezun, A., Ganesh, V., Guo, P.J., Hooimeijer, P., Ernst, M.D.: HAMPI: a solver for string constraints. In: Rothermel, G., Dillon, L.K. (eds.) Intl. Symp. on Software Testing and Analysis (ISSTA), pp. 105–116. ACM, New York (2009)
103. Kim, H., Somenzi, F.: Finite instantiations for integer difference logic. In: Formal Methods in Computer Aided Design (FMCAD), pp. 31–38. IEEE, Piscataway (2006)
104. King, T., Barrett, C., Dutertre, B.: Sum of infeasibility simplex for SMT. In: Formal Methods in Computer Aided Design (FMCAD), pp. 189–196. IEEE, Piscataway (2013)
105. Korovin, K.: iProver—an instantiation-based theorem prover for first-order logic (system description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) Intl. Joint Conf. on Automated Reasoning (IJCAR). LNCS, vol. 5195, pp. 292–298. Springer, Heidelberg (2008)
106. Korovin, K., Tsiskaridze, N., Voronkov, A.: Conflict resolution. In: Gent, I. (ed.) Intl. Conf. on Principles and Practice of Constraint Programming (CP). LNCS, vol. 5732, pp. 509–523. Springer, Heidelberg (2009)
107. Kroening, D., Strichman, O.: Decision Procedures: An Algorithmic Point of View. Springer, Heidelberg (2008)
108. Krstić, S., Goel, A.: Architecting solvers for SAT modulo theories: Nelson-Oppen with DPLL. In: Konev, B., Wolter, F. (eds.) Intl. Symp. on Frontiers of Combining Systems (FroCoS). LNCS, vol. 4720, pp. 1–27. Springer, Heidelberg (2007)
109. Krstić, S., Goel, A., Grundy, J., Tinelli, C.: Combined satisfiability modulo parametric theories. In: Grumberg, O., Huth, M. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 4424, pp. 618–631. Springer, Heidelberg (2007)
110. Lahiri, S.K., Seshia, S.A., Bryant, R.E.: Modeling and verification of out-of-order microprocessors in UCLID. In: Aagaard, M., O'Leary, J.W. (eds.) Formal Methods in Computer Aided Design (FMCAD). LNCS, vol. 2517, pp. 142–159. Springer, Heidelberg (2002)
111. Lev-Ami, T., Immerman, N., Reps, T., Sagiv, M., Srivastava, S., Yorsh, G.: Simulating reachability using first-order logic with applications to verification of linked data structures. In: Nieuwenhuis, R. (ed.) Intl. Conf. on Automated Deduction (CADE). LNCS, vol. 3632, pp. 99–115. Springer, Heidelberg (2005)

112. Manna, Z., Zarba, C.G.: Combining decision procedures. In: Aichernig, B.K., Maibaum, T. (eds.) Formal Methods at the Crossroads: From Panacea to Foundational Support. LNCS, vol. 2757, pp. 381–422. Springer, Heidelberg (2003)
113. Manolios, P., Srinivasan, S., Vroon, D.B.: The bit-level analysis tool. In: Damm, W., Hermanns, H. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 4590, pp. 303–306. Springer, Heidelberg (2007)
114. Manzano, M.: Introduction to many-sorted logic. In: Meinke, K., Tucker, J.V. (eds.) Many-Sorted Logic and Its Applications, pp. 3–86. Wiley, New York (1993)
115. McMillan, K.: Interpolation and SAT-based model checking. In: Hunt, W.A. Jr., Somenzi, F. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 2725, pp. 1–13. Springer, Heidelberg (2003)
116. McMillan, K.L.: Applications of Craig interpolants in model checking. In: Halbwachs, N., Zuck, L.D. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 3440, pp. 1–12. Springer, Heidelberg (2005)
117. McMillan, K.L.: An interpolating theorem prover. Theor. Comput. Sci. **345**(1), 101–121 (2005)
118. McMillan, K.L.: Interpolation and model checking. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
119. McMillan, K.L., Kuehlmann, A., Sagiv, M.: Generalizing DPLL to richer logics. In: Bouajjani, A., Maler, O. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 5643, pp. 462–476 (2009)
120. Moskal, M.: Rocket-fast proof checking for SMT solvers. In: Ramakrishnan, C.R., Rehof, J. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 4963, pp. 486–500. Springer, Heidelberg (2008)
121. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
122. de Moura, L., Bjørner, N.: Generalized, efficient array decision procedures. In: Formal Methods in Computer Aided Design (FMCAD), pp. 45–52. IEEE, Piscataway (2009)
123. de Moura, L., Bjørner, N.S.: Model-based theory combination. In: Intl. Workshop on Satisfiability Modulo Theories (SMT) (2007)
124. de Moura, L., Rueß, H.: Lemmas on demand for satisfiability solvers. In: Intl. Symp. on the Theory and Applications of Satisfiability Testing (SAT) (2002)
125. de Moura, L., Rueß, H., Sorea, M.: Bounded model checking and induction: from refutation to verification. In: Hunt, W.A. Jr., Somenzi, F. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 2725, pp. 14–26. Springer, Heidelberg (2003)
126. Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. Trans. Program. Lang. Syst. **1**(2), 245–257 (1979)
127. Nelson, G., Oppen, D.C.: Fast decision procedures based on congruence closure. J. ACM **27**(2), 356–364 (1980)
128. Nieuwenhuis, R., Oliveras, A.: DPLL(T) with exhaustive theory propagation and its application to difference logic. In: Etessami, K., Rajamani, S.K. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 3576, pp. 321–334. Springer, Heidelberg (2005)
129. Nieuwenhuis, R., Oliveras, A.: On SAT modulo theories and optimization problems. In: Biere, A., Gomes, C.P. (eds.) Theory and Applications of Satisfiability Testing (SAT). LNCS, vol. 4121, pp. 156–169. Springer, Heidelberg (2006)
130. Nieuwenhuis, R., Oliveras, A.: Fast congruence closure and extensions. Inf. Comput. **205**(4), 557–580 (2007)
131. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Abstract DPLL and abstract DPLL modulo theories. In: Baader, F., Voronkov, A. (eds.) Intl. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR). LNCS, vol. 3452, pp. 36–50. Springer, Heidelberg (2005)
132. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories: from an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). J. ACM **53**(6), 937–977 (2006)

133. Oe, D., Reynolds, A., Stump, A.: Fast and flexible proof checking for SMT. In: Intl. Workshop on Satisfiability Modulo Theories (SMT) (2009)
134. Oppen, D.C.: Complexity, convexity and combinations of theories. Theor. Comput. Sci. **12**, 291–302 (1980)
135. Peleska, J., Vorobev, E., Lapschies, F.: Automated test case generation with SMT-solving and abstract interpretation. In: Bobaru, M.G., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NASA Formal Methods Symp. (NFM). LNCS, vol. 6617, pp. 298–312. Springer, Heidelberg (2011)
136. Piskac, R., Kuncak, V.: Decision procedures for multisets with cardinality constraints. In: Logozzo, F., Peled, D., Zuck, L.D. (eds.) Intl. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI), vol. 4905, pp. 218–232. Springer, Heidelberg (2008)
137. Pnueli, A., Rodeh, Y., Shtrichman, O., Siegel, M.: Deciding equality formulas by small domains instantiations. In: Halbwachs, N., Peled, D. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 1633, pp. 687–688. Springer, Heidelberg (1999)
138. Presburger, M.: Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In: Comptes Rendus du I Congrès de Mathématiciens des Pays Slaves, Warsaw, pp. 92–101 (1929)
139. Pudlák, P.: Lower bounds for resolution and cutting planes proofs and monotone computations. J. Symb. Log. **62**(3), 981–998 (1997)
140. Pugh, W.: The omega test: a fast and practical integer programming algorithm for dependence analysis. In: Martin, J.L. (ed.) Conf. on Supercomputing (SC), pp. 4–13. IEEE/ACM, Piscataway/New York (1991)
141. Ranise, S., Ringeissen, C., Zarba, C.G.: Combining data structures with nonstably infinite theories using many-sorted logic. In: Gramlich, B. (ed.) Intl. Symp. on Frontiers of Combining Systems (FroCoS). LNCS, vol. 3717, pp. 48–64. Springer, Heidelberg (2005)
142. Reynolds, A., Tinelli, C., Goel, A., Krstić, S.: Finite model finding in SMT. In: Sharygina, N., Veith, H. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 8044, pp. 640–655. Springer, Heidelberg (2013)
143. Reynolds, A., Tinelli, C., Goel, A., Krstić, S.: Quantifier instantiation techniques for finite model finding in SMT. In: Bonacina, M.P. (ed.) Intl. Conf. on Automated Deduction (CADE). LNCS, vol. 7898, pp. 377–391. Springer, Heidelberg (2013)
144. Rümmer, P., Wahl, T.: An SMT-LIB theory of binary floating-point arithmetic. In: Intl. Workshop on Satisfiability Modulo Theories (SMT) (2010)
145. Rybalchenko, A., Sofronie-Stokkermans, V.: Constraint solving for interpolation. J. Symb. Comput. **45**, 1212–1233 (2010)
146. Schrijver, A.: Theory of Linear and Integer Programming. Wiley, New York (1986)
147. Sebastiani, R.: Lazy satisfiability modulo theories. J. Satisf. Boolean Model. Comput. **3**(3–4), 141–224 (2007)
148. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In: Hunt, W.A. Jr., Johnson, S.D. (eds.) Formal Methods in Computer Aided Design (FMCAD). LNCS, vol. 1954, pp. 108–125. Springer, Heidelberg (2000)
149. Sheini, H.M., Sakallah, K.A.: A scalable method for solving satisfiability of integer linear arithmetic logic. In: Bacchus, F., Walsh, T. (eds.) Theory and Applications of Satisfiability Testing (SAT). LNCS, vol. 3569, pp. 241–256. Springer, Heidelberg (2005)
150. Sofronie-Stokkermans, V.: Hierarchic reasoning in local theory extensions. In: Nieuwenhuis, R. (ed.) Intl. Conf. on Automated Deduction (CADE). LNCS, vol. 3632, pp. 219–234. Springer, Heidelberg (2005)
151. Strichman, O.: On solving Presburger and linear arithmetic with SAT. In: Aagaard, M., O'Leary, J.W. (eds.) Formal Methods in Computer Aided Design (FMCAD). LNCS, vol. 2517, pp. 160–170. Springer, Heidelberg (2002)
152. Strichman, O., Seshia, S., Bryant, R.: Deciding separation formulas with SAT. In: Brinksma, E., Larsen, K.G. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 2404, pp. 113–124. Springer, Heidelberg (2002)

153. Stump, A., Barrett, C.W., Dill, D.L., Levitt, J.: A decision procedure for an extensional theory of arrays. In: Symp. on Logic in Computer Science, vol. LICS, pp. 29–37. IEEE, Piscataway (2001)
154. Stump, A., Oe, D., Reynolds, A., Hadarean, L., Tinelli, C.: SMT proof checking using a logical framework. Form. Methods Syst. Des. **41**(1), 91–118 (2013)
155. Tarjan, R.E.: Efficiency of a good but not linear set union algorithm. J. ACM **22**(2), 215–225 (1975)
156. Tinelli, C.: A DPLL-based calculus for ground satisfiability modulo theories. In: Ianni, G., Flesca, S. (eds.) European Conf. on Logics in Artificial Intelligence (JELIA). LNCS, vol. 2424, pp. 308–319. Springer, Heidelberg (2002)
157. Tinelli, C., Harandi, M.T.: A new correctness proof of the Nelson-Oppen combination procedure. In: Baader, F., Schulz, K.U. (eds.) Intl. Workshop on Frontiers of Combining Systems (FroCoS), pp. 103–120. Kluwer Academic, Dordrecht (1996)
158. Tinelli, C., Ringeissen, C.: Unions of non-disjoint theories and combinations of satisfiability procedures. Theor. Comput. Sci. **290**(1), 291–353 (2003)
159. Tinelli, C., Zarba, C.: Combining decision procedures for sorted theories. In: Alferes, J.J., Leite, J.A. (eds.) European Conf. on Logics in Artificial Intelligence (JELIA). Lecture Notes in Artificial Intelligence, vol. 3229, pp. 641–653. Springer, Heidelberg (2004)
160. Tinelli, C., Zarba, C.: Combining nonstably infinite theories. J. Autom. Reason. **34**(3), 209–238 (2005)
161. Wang, C., Gupta, A., Ganai, M.: Predicate learning and selective theory deduction for a difference logic solver. In: Sentovich, E. (ed.) Design Automation Conf. (DAC), pp. 235–240. ACM, New York (2006)
162. Yorsh, G., Musuvathi, M.: A combination method for generating interpolants. In: Nieuwenhuis, R. (ed.) Intl. Conf. on Automated Deduction (CADE). LNCS, vol. 3632, pp. 353–368. Springer, Heidelberg (2005)

Chapter 12
Compositional Reasoning

Dimitra Giannakopoulou, Kedar S. Namjoshi, and Corina S. Păsăreanu

Abstract State Explosion is a fundamental challenge for model checking methods. This term refers to the potentially exponential growth of the state space of a program as a function of the number of its components. Compositional reasoning is a technique which aims to ameliorate the effects of state explosion. In its essence, it replaces reasoning on the global state space of a program with localized reasoning: each component is analyzed separately, based on assumptions about the behavior of the other components. The challenge for a fully automated method is the construction of the right assumptions: they should be strong enough to prove a desired property, while being simple enough for efficient analysis. This chapter describes the ideas underlying compositional reasoning, foundational algorithms for generating assumptions, and applications.

12.1 Introduction

Concurrent programs are difficult to analyze. Informally, this is because any proof of correctness must keep track of multiple concurrently active threads of control. This informal view can be crisply formalized as the question of whether a specified global state of a program with N concurrently active components is reachable. This question is PSPACE-hard in N. One proof goes by a reduction from the IN-PLACE-ACCEPTANCE problem [77], by letting each component simulate a single tape cell; hardness holds even if the state space of every component is a constant independent of N. In practice, the difficulty manifests itself as an exponential growth (in N) of the number of program states, often referred to as "state explosion". Exponential growth makes it difficult to analyze programs with standard model-checking techniques, such as those which are based on computing the reachable state space.

D. Giannakopoulou · C.S. Păsăreanu
NASA Ames Research Center, Moffett Field, CA, USA

K.S. Namjoshi (✉)
Bell Labs, Nokia, Murray Hill, NJ, USA
e-mail: kedar.namjoshi@nokia-bell-labs.com

E.M. Clarke et al. (eds.), *Handbook of Model Checking*,
DOI 10.1007/978-3-319-10575-8_12

This chapter explores an alternative reasoning principle called "compositional reasoning". (Other common names are "local", "modular", "assume-guarantee", "assumption-commitment" and "rely-guarantee".) The essence of this principle is to replace a single analysis over the global state space with a number of localized analyses. A local analysis examines a single component, abstracting the rest of the program as an *assumption* for this component. A compositional proof rule is set up to ensure mutual consistency among the per-component assumptions. The challenge in applying such a rule is the construction of proper assumptions: they should be strong enough to prove a desired property, while being amenable to simple analysis. This chapter describes the ideas underlying compositional reasoning, foundational algorithms for generating assumptions, and applications. Our focus is on the algorithms for compositional verification and assumption generation; for a deeper discussion of the underlying proof principles, the book [83] is an excellent reference.

PSPACE-hardness of the verification question implies that compositional reasoning cannot avoid state explosion for *all* programs (supposing P $\neq$ PSPACE). On the other hand, many programs can be viewed as "loosely coupled" collections of components, where the behavior of a component is influenced only to a limited degree by the behavior of the others. For such programs, one may expect localized reasoning to perform significantly better than a global analysis.

For sequential programs, compositional reasoning is built into Hoare's proof rules for **while** programs [53] and the denotational semantics of Scott and Strachey [84]. Both were developed in the late 1960s. For concurrent programs, the seminal work on compositional reasoning principles is that of Owicki and Gries and Lamport, from the mid-1970s [60, 76]. This work inspired the creation of fully compositional methods by Misra and Chandy [69] and by Jones [57]. In these methods, each process is associated with an *assumption* on its input and a *guarantee* on its output. The proof rules ensure mutual consistency of the assumptions and guarantees. The methods are known as "assume-guarantee methods" for this reason. We use "A-G reasoning" as an abbreviation.

We consider two models of concurrency. In both, the execution of components is asynchronous or loosely synchronized. In one model, components communicate via shared memory; in the other, communication is through synchronized (buffer-less) message exchange. The focus of the chapter is on algorithms which automatically construct assumptions with which to instantiate the compositional rules. We describe only briefly the issues that arise in the design of correct compositional rules, but give references for further reading. References are also given to compositional rules for other models of concurrency, in particular for the important model of fully synchronized (hardware) processes.

There is a variety of assume-guarantee rules in the literature, but they can be understood using only two core principles. These can be explained informally in terms of the notation $\{a\}M\{g\}$, which indicates that component M guarantees a property g under assumption a. (A rough interpretation is as the implication $[(a \wedge M) \Rightarrow g]$; the following sections have precise definitions.)

The simplest principle is that of *transitivity*. For example, suppose that component M satisfies $\{a\}M\{g\}$ while component N satisfies $\{true\}N\{a\}$. Reasoning

based on the transitivity of implication establishes that the composition $N//M$ satisfies $\{true\}N//M\{g\}$. This form of reasoning applies when assumptions and guarantees are related in an acyclic manner.

The second principle is that of *mutual induction*. This applies if assumptions and guarantees are related in an (apparently) circular manner. Suppose that M satisfies $\{a\}M\{g\}$ while N satisfies $\{g\}N\{a\}$. Here, the assumptions and guarantees are circular: the guarantee of one process forms the assumption of the other. Hence, one can no longer use transitivity to show that $N//M$ guarantees g. Soundness arguments for circular rules typically rely on mutual induction over an appropriate well-founded domain, such as the length of finite computations. Therefore, special care must be taken with properties which are naturally defined over infinite computations, such as properties which incorporate liveness and fairness. In particular, circular rules that are sound for safety properties may not be sound for liveness properties.

A-G reasoning is a particular form of process abstraction. It is special in that it makes use of the internal structure of a process, in particular the connectivity between its components. Abstraction can be seen as operating on two levels. First, the influence of one process on another is defined solely in terms of the interface between the processes, so that much of the internal structure of a process can be ignored. Second, the behavior of a process is directly influenced only by its immediate neighbors; this abstracts away processes which are further off within a process network.

This chapter describes two types of rules for assume-guarantee reasoning. The first kind operate at the level of proof outlines (assertions on state, ranking functions for termination) as seen in the work of Owicki–Gries, Lamport, and Jones, referred to earlier. These rules are considered in Sect. 12.2. A second type of rule is based on the semantics of a process as a language of program traces, as seen in the work of Misra–Chandy and Pnueli [80]. Such rules are considered in Sect. 12.3.

The goal of an automated method for compositional analysis is to find the 'right' assumptions and guarantees for each component, in order to prove a global property for the composite program. Experience has shown (cf. [61]) that designing such abstractions by hand can be quite difficult: an abstraction must be strong enough to prove global correctness properties, while also being simple enough for efficient analysis. The automated methods discussed here take an iterative approach to the construction of assumptions, progressing from simpler assumptions to more complex ones. They can be viewed as algorithms which "learn" the assumptions required to correctly instantiate a compositional rule. Learning is an iterative process: in each iteration, counterexamples to the currently conjectured assumptions and guarantees are used to refine the conjecture.

The following sections define compositional rules and their associated algorithms. Section 12.2 lays out a shared-variable communication model and compositional rules which are based on invariants and ranking functions. Section 12.3 lays out a model based on process communication and compositional rules which are based on the semantics of a process as a set of execution traces. We also collect and organize a number of references for further reading, including pointers to work on compositional rules for other models of concurrent execution.

12.2 Reasoning with Assertions

In this section, we examine assertion-based compositional reasoning methods. These methods derive from the seminal work in the mid-1970s by Susan Owicki and David Gries [76] and independently by Leslie Lamport [60]. It is usual to refer to the original method as the Owicki–Gries method.

The Owicki–Gries method is an attempt to extend to concurrent programs the compositional reasoning rules developed by Hoare for sequential programs. From that point of view, the method is only a partial success as it is localized but not compositional. Historically, however, further work led to fully compositional assume-guarantee rules for concurrency. These rules lead to automatic methods of computing consistent assumptions and guarantees, as described in this section.

12.2.1 The (Non-compositional) Owicki–Gries Method

For simplicity, we consider a program M with two components M_1, M_2. The components represent independent, asynchronous threads of execution. We denote their combination as $M = M_1//M_2$. One may attempt to extend the compositional style of Hoare's method to this new operator by the following proof rule: if Hoare triples $\{P_1\}M_1\{Q_1\}$ and $\{P_2\}M_2\{Q_2\}$ hold for the individual components, the triple $\{P_1 \wedge P_2\}M\{Q_1 \wedge Q_2\}$ holds for the composition. This rule is unsound, though, as shown by the example below.

```
var x: integer; initially x=0

M1:: l: {x >= 0} x := 1 k: {x = 1}
M2:: l: {x >= 0} x := 2 k: {x = 2}
```

The individual proof outlines are correct, but the conclusion of the proof rule, which is $\{x \geq 0\}M\{\mathit{false}\}$, is not! Operationally, the conclusion implies that the program M does not terminate from a state where $x = 0$. An alternative view is that it requires the predicate transformer for M to be "miraculous" (cf. Dijkstra [31]).

Owicki and Gries traced the failure of this rule to the "interference" (their term) of the actions of M_2 with the proof outline of M_1. Operationally, the statement `x := 2` of M_2 may execute after the statement `x := 1` of M_1, changing the value of `x` to 2; by doing so, it causes a failure of the proof assertion `{x=1}` in the outline for M_1.

The less straightforward but correct formulation of the proof rule adds a side condition, called *non-interference*: no statement of M_2 should change an assertion in the proof outline of M_1, and vice-versa. The modified proof outline given below is free of interference, and shows that the composed program satisfies the triple $\{x \geq 0\}M_1//M_2\{x \geq 1\}$. Note that this outline has weaker assertions than the original; weakening is usually required for interference-freedom.

```
var x: integer; initially x=0

M1:: l: {x >= 0} x := 1 k: {x >= 1}
M2:: l: {x >= 0} x := 2 k: {x >= 1}
```

Formally, a statement S with pre-condition P in a proof outline for M_1 *does not interfere* with an assertion A in a proof outline for M_2 if the Hoare triple $\{P \wedge A\}S\{A\}$ holds. Informally, one may say that the transition S from any state satisfying $(P \wedge A)$ does not falsify A. If each proof outline has N statements and assertions, checking non-interference is a task of complexity $O(N^2)$.

As shown by the example, it can be difficult to discover a set of assertions which provide an interference-free proof. For these reasons, hand-constructing Owicki–Gries style proofs is not to be recommended. (Lamport even goes so far as to title one of his papers [61] "*Composition: A way to make proofs harder*"!) We show that these objections can be removed by automating the task of obtaining an interference-free proof.

Although the Owicki–Gries method was created with the goal of compositional reasoning, it is not compositional in the strict sense. A generally accepted definition of a compositional method is one where, assuming every component P_i satisfies the property φ_i, the composition $P = (//\, i \in [1, n] : P_i)$ satisfies the property $F(\varphi_1, \ldots, \varphi_n)$, for a specific function F. In the Owicki–Gries method, the final Hoare triple cannot be defined solely from the Hoare triples for each component; it is necessary to add the non-interference side condition which examines the proof outlines for the Hoare triples. This makes the method non-compositional.

On the other hand, if each property φ_i is a proof outline (vs. a triple), the non-interference check can be "built in" to the function F, so one has a truly compositional proof rule. Indeed, this is precisely how the compositional form of the Owicki–Gries rule is constructed. Proof outlines are somewhat unwieldy objects, so instead one uses an equivalent representation as an invariant assertion. This requires the introduction of an auxiliary variable, π, which denotes the program location. For example, the proof outline given above for M_1 is turned into the assertion $(\pi_1 = l \Rightarrow x \geq 0) \wedge (\pi_1 = k \Rightarrow x \geq 1)$, which is an invariant for M_1. The next section describes more precisely how the semantic, program invariant view gives rise to a compositional proof rule and to a fixpoint method for calculating interference-free proofs.

12.2.2 The Assume-Guarantee View: Localized Inductive Invariants

The explanation of the Owicki–Gries method given previously is in terms of proof outlines, which is tied to the syntax of a particular programming language. In this section, we examine a semantic view based on localized invariants. Besides being

compositional, the semantic view is simpler and forms a better foundation for automated algorithms and for extensions of the method beyond safety properties. We begin with preliminaries and necessary notation.

12.2.2.1 The Shared-Variable Program Model

A *program* is given by a tuple (V, I, T), where V is a finite set of program variables; $I(V)$ defines a set of initial states; and $T(V, V')$ defines a successor relation, with V' an isomorphic copy of V. The underlying state space of the program is given by the set of assignments of values to variables. The *strongest post-condition* predicate transformer $sp(T, \xi)$ defines the set of states which are successors under T of the states satisfying ξ. (It is also known as *post*.) Formally, the transformer is defined by $sp(T, \xi)(V) = (\exists V : T(V, V') \wedge \xi(V))\langle x : x \in V : x' := x\rangle$. The angled brackets define a substitution operator which replaces each variable x' with its corresponding unprimed variable x.

Consider a program M with components $\{M_k\}$ and let M_k be given by (V_k, I_k, T_k). The program M formed by *asynchronous composition* is defined by the tuple (V, I, T), where the set of variables V is the union $(\cup k : V_k)$ of the component variables, and the initial condition I is the conjunction $(\wedge k : I_k)$ of the initial conditions of the components. Every transition of M is a transition by some component, which leaves the values of all variables of other components unchanged. Formally, the joint transition relation T is defined by $(\vee k : T_k(V_k, V_k') \wedge (\forall j : j \neq k : unch(V_j \backslash V_k)))$, where $unch(W)$, read as "W is unchanged", is given by the formula $(\wedge w : w \in W : w' = w)$.

A Note on Notation

We use a succinct notation introduced by Dijkstra and Scholten [30]. The bracketing operator $[\Phi]$ denotes universal quantification over all free variables in Φ. Thus, $[I \Rightarrow \xi]$ is short for $(\forall V : I(V) \Rightarrow \xi(V))$, which itself is another way of saying that the set of states satisfying I is a subset of those satisfying ξ.

Invariant Assertions

An *assertion* or *predicate* is a Boolean-valued function on program states. An assertion $\xi(V)$ defines the set of states where ξ evaluates to *true*. The fundamental notion in assertion-based proofs of safety is that of an *inductive invariant*. An assertion ξ is an *inductive invariant* if it includes all initial states, i.e., $[I \Rightarrow \xi]$, and it is *inductive*, i.e., it is closed under program transitions, written $[sp(T, \xi) \Rightarrow \xi]$. A standard proof rule for showing that an assertion φ is invariant is to find an inductive invariant ξ such that $[\xi \Rightarrow \varphi]$ holds.

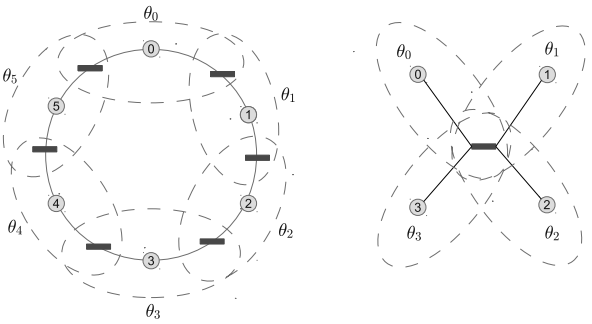


Fig. 1 The scope of a split invariant, in a ring topology (*left*) and a centralized star topology (*right*). *Circles* represent process nodes, *rectangles* represent shared memory

12.2.2.2 Split Invariants

An assertion is *local* to a component M_k if it is defined only over the variables V_k which belong to M_k. A *separable* assertion is one which is a Boolean combination of local assertions. A conjunctive separable assertion has the form $\theta_1(V_1) \wedge \theta_2(V_2) \wedge \cdots \wedge \theta_n(V_n)$, one for each component $M_1, \ldots, M_n$. It can be represented as a vector $\theta = (\theta_1, \theta_2, \ldots, \theta_n)$ of local assertions, one per component. We call such an assertion a *split invariant* if the conjunction is globally inductive. Figure 1 illustrates the local scope of the components of a split invariant, for the ring and star topologies.

Definition 1 A split invariant is a conjunctively separable assertion which is a global inductive invariant.

For simplicity, we consider a two-component system, $M = M_1 // M_2$ where the components communicate using a set of shared variables, X, defined by $X = V_1 \cap V_2$. We let $L_i = V_i \backslash X$, so that L_1, L_2, and X are mutually disjoint. Let $\theta = \theta_1(V_1) \wedge \theta_2(V_2)$ be a split invariant for that system. We simplify the initiality and inductiveness conditions using the locality of predicates and show how the non-interference condition emerges naturally from this exercise.

The initial condition is $[I \Rightarrow (\theta_1 \wedge \theta_2)]$, which is equivalent to the pair of conditions below, for $i \in [1, 2]$.

$$[I \Rightarrow \theta_i]. \tag{1}$$

The inductiveness condition is $[sp(T, \theta_1 \wedge \theta_2) \Rightarrow (\theta_1 \wedge \theta_2)]$. By the definition of T and the distributivity of sp over $\vee$, this is equivalent to the four conditions below for $i, j \in [1, 2]$ and $i \neq j$.

$$\big[sp\big(T_i \wedge unch(L_j), \theta_i \wedge \theta_j\big) \Rightarrow \theta_i\big], \tag{2}$$

$$\big[sp\big(T_i \wedge unch(L_j), \theta_i \wedge \theta_j\big) \Rightarrow \theta_j\big]. \tag{3}$$

Condition (2) simplifies to

$$\big[sp_i\big(T_i, \theta_i \wedge (\exists L_j : \theta_j)\big) \Rightarrow \theta_i\big]. \tag{4}$$

The notation $sp_k(T_k, \xi)$ is used in place of $sp(T, \xi)$ when the support for ξ is limited to V_k. The formula (4) shows "near-inductiveness" of θ_i. It is weaker than inductiveness due to the additional pre-condition $(\exists L_j : \theta_j)$.

Condition (3) simplifies to the following

$$\left[sp_j\left(\left(\exists L_i', L_i : T_i \wedge \theta_i\right) \wedge unch(L_j), \theta_j\right) \Rightarrow \theta_j\right]. \tag{5}$$

Informally, this requires that actions of process M_i leave θ_j unchanged. This is precisely the non-interference requirement, now expressed semantically in terms of invariance predicates rather than proof outlines!

We use $sum_i(\theta)$ (read as "summary for j") as an abbreviation for the term $(\exists L_i, L_i' : T_i \wedge \theta_i)$ which appears in Eq. (5). This formula is a transition term over X and X'. It may be thought of as a *summary* transition representing the interference due to transitions of M_i on the state of M_j.

A proof outline for a program is a mapping Q from program points to assertions. An outline is valid if whenever there is a transition S from program point p to program point q, the condition $[Q(p) \Rightarrow wlp(S, Q(q))]$ holds. A proof outline Q induces the assertion $\xi(Q) = (\wedge p : \pi = p \Rightarrow Q(p))$, where π is the program counter and the quantification is over program points p. The validity condition for Q implies that the assertion $\xi(Q)$ is an inductive invariant of the program. Viewed through this transformation, conditions (5) are precisely the Owicki–Gries non-interference assertions, while conditions (4) are slightly weaker forms of the local Hoare triples from the outline. The following theorem arises from this correspondence.

Theorem 1 *Any Owicki–Gries proof defines a split invariant. Conversely, any split invariant corresponds to an Owicki–Gries proof with slightly weaker inductiveness requirements.*

12.2.3 Computing the Strongest Split Invariant

Model-checking algorithms are usually based on the computation of fixpoints. The split invariance conditions also have a pre-fixpoint form, which is apparent when they are simplified (as described before) and considered together.

These implications in Eqs. (1), (4), and (5), gathered together, form a system of *simultaneous* implications. These have the general form $[F_i(\theta_i, \theta_j) \Rightarrow \theta_i]$, where F_i is a monotonic function on the lattice of assertion vectors ordered by componentwise implication.

Recall that, for simplicity, the program has only two components, so the indices i, j are distinct and range over $\{1, 2\}$. The specific function F_i is given by the disjunction below.

$$F_i = (\exists L_j : I) \vee sp_i\left(T_i, \theta_i \wedge (\exists L_j : \theta_j)\right) \vee sp_i\left(sum_j(\theta) \wedge unch(L_i), \theta_i\right). \tag{6}$$

Monotonicity of F_i follows from the monotonicity of sp and that of the Boolean operators. The following theorems are a direct consequence of the rewriting and the Knaster–Tarski fixpoint theorem.

Theorem 2 *Any split invariant is a solution to the simultaneous system of implications* $[F_i(\theta) \Rightarrow \theta_i]$ *for all* i.

Theorem 3 *There is a strongest split invariant, which is defined by the simultaneous least fixpoint of* F.

By the Knaster–Tarski theorem, the vector function $F = (F_1, F_2)$ has a least fixpoint. Moreover, this fixpoint can be calculated by the approximation sequence $\bot, F(\bot), F^2(\bot), \ldots$, where $\bot$ denotes the vector $(false, false)$. It can be shown that any fixpoint of F defines local assertions: i.e., that $[(\exists L_j : \theta_i) \equiv \theta_i]$ holds for all $i, j : i \neq j$.

By Theorem 1, split invariants are more general than Owicki–Gries proofs. However, the *strongest* split invariant is an Owicki–Gries proof. One can show by induction on the fixpoint stages that $[\theta_i \Rightarrow (\exists L_j : \theta_j)]$ holds at each stage; thus, the "near-invariance" condition (2) turns into the standard invariance condition.

Theorem 4 *The strongest split invariant corresponds to an Owicki–Gries proof; in fact, the strongest such proof outline.*

Split Invariance for N Processes

The generalization for N processes, $N \geq 1$, follows the same pattern. Rather than give the generalized implications, we describe below the fixpoint calculation algorithm which is based on those rules. The algorithm computes a vector θ of N components through a simultaneous fixpoint iteration where, in the $(K+1)$th iteration, all components are updated based on their values at the Kth iteration.

1. The initial value of the N-vector, θ, is *false* for each component.
2. At stage $K+1$, all components are updated together. The update for component i sets the new value, $\theta^{K+1}(i)$, to the union ($\vee$) of one of the terms below with the previous value, $\theta^K(i)$.

[initial]	$(\exists L_i : I)$
[step]	$sp_i(T_i, \theta^K(i))$
[interference]	$sp_i(sum_j(\theta^K) \wedge unch(L_i), \theta^K(i))$, for $j \neq i$.

3. The computation terminates when all components have reached a fixpoint.

This is a non-deterministic algorithm. The *chaotic iteration theorem* from [27] allows much freedom in evaluating a simultaneous fixpoint. Each component of the fixpoint vector may be updated asynchronously, so long as the overall evaluation schedule is fair, in that no component or update step is neglected forever. For instance, a schedule may iterate the "step" update for component i until θ_i does not change (in effect, doing local reachability in M_i), and only then apply the interference update. Several optimizations are also possible, such as a frontier-based calculation based on changes to the θ's.

12.2.4 Relationship to Rely-Guarantee

Introducing fresh variables $\{R_i, G_i\}$ into (5), one obtains the implications below.

$$\left[sp_i(R_i, \theta_i) \Rightarrow \theta_i\right], \tag{7}$$

$$\left[\left(\exists L_i', L_i : T_i \wedge \theta_i\right) \Rightarrow G_i\right], \tag{8}$$

$$[G_j \Rightarrow R_i], \quad \text{for } j \neq i. \tag{9}$$

This set of implications, together with the initial conditions given by (1) and the local invariance given by (4), forms the "rely-guarantee" method introduced by Jones [57]. The variables R_i and G_i clearly represent transition terms. Implication (8) says that G_i is weaker than the summary transition for M_i; it is therefore called the "guarantee" of M_i. The interference of all other processes on M_i is represented by the variable R_i. Informally, one may say that M_i "relies" upon the fact that interference is limited to R_i. The implication (7) says that the interference leaves θ_i invariant. The rely and guarantee terms must be linked up: the implication (9) says that the rely term for M_i is weaker than the guarantees provided by all other components.

Given a split invariant, one can define $G_i = (\exists L_i', L_i : T_i \wedge \theta_i)$ and $R_i = (\vee j : j \neq i : G_j \wedge unch(L_i))$, which meet the rely-guarantee conditions. Conversely, the strongest rely-guarantee proof is the least solution of the implications above, which is a fixpoint by the Knaster–Tarski theorem, so it has this shape for G_i and R_i. This connection is summarized below.

Theorem 5 *Any split invariant induces a rely-guarantee proof. Moreover, the strongest rely-guarantee proof is equivalent to the strongest split invariance proof.*

12.2.5 Completeness Issues

The standard inductive invariant proof rule is *complete*. This means that if an assertion φ is invariant, there is an *inductive* invariant, ψ, which is stronger than φ. Hence, any invariant can be proved by defining a stronger inductive invariant. The witness for completeness is trivial: the set of reachable states forms an inductive invariant, and it is (by definition) stronger than any other invariant assertion.

On the other hand, for the split invariants constructed in the Owicki–Gries method, completeness is not guaranteed. This can be shown by the following mutual exclusion protocol. The labels represent states: thinking (T), hungry (H), and eating (E). The transition from hungry to eating occurs if x is true. The transition uses an atomic test-and-set primitive, expressed as $< . . >$. This tests the associated condition and sets x to false in a single action. The transition from eating to thinking re-sets x to true.

```
var x: boolean; initially x=true

M1:                                M2:
  while (true) {                     while (true) {
  T: skip;                           T: skip;
  H: <if x then x:= false>           H: <if x then x:= false>
  E: x := true                       E: x := true
  }                                  }
```

This program satisfies the mutual exclusion property: there is no reachable state of the program where `M1` and `M2` are both in location E. However, the least fixpoint computation of the strongest split invariant produces the result $(\theta_1, \theta_2) = (\mathit{true}, \mathit{true})$. While the conjunction, *true*, is a global invariant, it does not imply mutual exclusion.

Owicki–Gries and Lamport recognized this problem, and showed that completeness could be regained by the addition of *shared* auxiliary variables to the program. An auxiliary variable is a fresh shared variable that is updated with the original program, but does not influence program behavior. As an auxiliary variable is shared, it appears in the support of each split invariant component, and thus indirectly tightens the constraints between the local state of different components. As an example, if w is an auxiliary shared variable, and l_1, l_2 are local variables for components M_1 and M_2 respectively, the split assertion $(w \geq l_1) \wedge (l_2 \geq w)$ implies the joint assertion $(l_2 \geq l_1)$, which cannot be expressed by the individual invariants.

Owicki and Gries showed that adding a single variable which records the history of actions in an execution suffices for completeness. Lamport proposed a different approach, which exposes portions of the local state of components through auxiliary variables. History information, while unbounded in the general case, need not always be so. For instance, in the example above, adding a history variable "last", which records the last process to enter the critical region, suffices to compute a strong invariant. The program with the changes due to the auxiliary variable is shown below.

```
var x: boolean; initially x=true
var last: {1,2}; initially last=1

M1:                                M2:
  while (true) {                     while (true) {
  T: skip;                           T: skip;
  H: <if x then                      H: <if x then
      {x:=false;last:=1}>                {x:= false;last:=2}>
  E: x := true                       E: x := true
  }                                  }
```

The fixpoint computation on this program results in the solution $\theta(i) = (@E(i) \equiv \neg x \wedge (\mathit{last} = i))$, where $@E(i)$ is a predicate true when the program counter of component M_i is at location E. This suffices to show mutual exclusion:

if there is a reachable state where both components are at location E, the invariant implies that $(last = 1)$ and $(last = 2)$ must both be true, a contradiction.

12.2.6 Deadlock Detection with Local Invariants

A program enters a deadlock state when no transition is possible by any of its components. Thus, absence of deadlock can be formulated as the invariance of the assertion $(\vee j : (\exists L'_j : T_j))$. This can be shown through a split invariant. For the mutual exclusion example, the only state where a deadlock might occur is one in which both components are in state H and x is false. The stronger split invariant computed above implies that in this state, *last* is neither 1 nor 2. This is a contradiction, as the variable *last* must have one of these two values by its definition. Hence, the split invariant suffices to show absence of deadlock.

12.2.7 Local Proofs for Termination, Temporal Properties, and Fairness

We extend the split invariance calculation to show termination and other temporal liveness properties, including those that hold only under fairness assumptions.

12.2.7.1 Background

Assertional proof methods for termination combine an inductive invariant with a ranking argument. A ranking function, ρ, is a partial function which maps a program state to its "rank", an element of a well-founded set $(W, \prec)$. In the context of an inductive invariant, θ, this function must satisfy two conditions:

1. The function ρ is defined for all states where the invariant θ holds. Formally, $[\theta \Rightarrow domain(\rho)]$, and
2. The value of ρ must decrease strictly across every program transition from an invariance state. Formally, $[\theta \wedge (\rho = k) \Rightarrow wlp(T, \rho \prec k)]$, for every k in W.

Termination is implied, as follows. Consider a non-terminating execution, σ. Then σ is infinite, and every state on σ satisfies the invariant, θ. Hence, ρ is defined at each state of σ. From the second condition, the values taken by ρ on σ form an infinite strictly decreasing sequence. This contradicts the well-foundedness of the rank domain.

Ranking functions are also a key component of proof rules for temporal liveness properties and fairness properties. We consider a temporal property which is specified by a Büchi automaton for its negation. A Büchi automaton is specified as a tuple, $(Q, Q^0, \Sigma, \delta, G)$, where Q is a finite set of states, Q^0 is a set of initial states,

a non-empty subset of Q, δ is a transition relation, a subset of $Q \times \Sigma \times Q$, and G is a set of accepting, "green", states. A *run* of the automaton on an infinite sequence $\sigma : \mathbb{N} \to \Sigma$ is given by a function $r : \mathbb{N} \to Q$ such that $r(0)$ is in Q^0, and for each i, $(r(i), \sigma(i), r(i+1))$ is in δ. The run r is *accepting* iff there are infinitely many i such that $r(i)$ is in G. An accepting run is thus marked by an infinite sequence of green flashes.

A program M induces a transition system (S, I, T) where S is the set of program states, I is the initial set of states, and T is the transition relation. The *synchronous product* of a program M and an automaton A, written $M \times A$, is given by the automaton B with alphabet $\{\epsilon\}$ and $Q_B = S_M \times Q_A$, $Q_B^0 = I_M \times Q_A^0$, $G_B = S_M \times G_A$, and where $((s, q), \epsilon, (s', q')) \in \delta_B$ iff $(s, s') \in T_M$ and $(q, s, q') \in \delta_A$. If A specifies the *negation* of the desired temporal property φ, then M satisfies φ iff there is no accepting run of A on a computation of M, which holds iff there is no accepting run of B on the infinite sequence of ϵ-symbols.

For a correct program, a rank function is used to show that an accepting run does not exist. An invariant, θ, and a partial rank function, ρ, are supplied for the product $B = M \times A$. The rank function is constrained to satisfy three conditions:

1. The function ρ must be defined for all states satisfying the invariant; i.e., $[\theta \Rightarrow \mathit{domain}(\rho)]$,
2. The value of ρ cannot increase across any program transition from a state satisfying the invariant; i.e., $[\theta \wedge (\rho = k) \Rightarrow \mathit{wlp}(T_B, \rho \preceq k)]$, for all k in W, and
3. The value of ρ must strictly decrease across any transition from a green invariant state; i.e., $[\theta \wedge (\rho = k) \wedge \mathit{Green} \Rightarrow \mathit{wlp}(T_B, \rho \prec k)]$, for all k in W.

Establishing θ and ρ with these properties ensures correctness. Suppose, to the contrary, that the program is incorrect. Hence, there is an infinite accepting run, σ, of the automaton on a program computation, which corresponds to an infinite computation of $M \times A$ along which green states occur infinitely often. Every state of the run satisfies the invariant θ; hence, the rank ρ is defined for that state. By the second condition, the rank values do not increase along σ. By the third condition, the rank values must strictly decrease for infinitely many transitions on σ, inducing an infinite, strictly decreasing sequence of ranks, which contradicts the well-foundedness of the rank domain.

12.2.7.2 Local Proof Rules for Liveness Properties

Owicki and Gries developed a localized proof rule for termination which replaces the global rank function with per-component rank functions, each mapping to its own well-founded domain. Non-interference is extended to ranking functions: the rank of a (component) state cannot increase due to a transition from another component. This localized termination rule can be made complete by the introduction of auxiliary variables. We first describe a generalization of this rule to temporal properties, then show one way of treating fairness assumptions.

The localized rule for temporal properties restricts the property so that it is defined by a Büchi automaton which operates only on the shared state. The non-compositional proof rule given above for correctness is localized by replacing the global invariant with a split invariant, and the global ranking function with a split ranking function. For the ith component, let $B_i = M_i \times A$ be the synchronous product of that component with the automaton. A ranking function, ρ_i, is defined over B_i, and maps to a well-founded set $(W_i, \prec_i)$. In addition to the proof rules showing that θ is a split invariant, the rules for ranking functions are the following.

1. The function ρ_i must be defined for all states satisfying the invariant; i.e., $[\theta_i \Rightarrow domain(\rho_i)]$,
2. The value of ρ_i cannot increase across any program transition from a state satisfying the invariant; i.e., $[\theta_i \wedge (\rho_i = k) \Rightarrow wlp(T_{B_i}, \rho_i \preceq k)]$, for all k in W,
3. The value of ρ must strictly decrease across any transition from a green invariant state; i.e., $[\theta_i \wedge (\rho_i = k) \wedge Green_i \Rightarrow wlp(T_{B_i}, \rho_i \prec k)]$, for all k in W,
4. The value of ρ_i cannot increase across any interference transition; i.e., $[\theta_i \wedge (\rho_i = k) \Rightarrow wlp(sum_j(\theta) \wedge unch(L_i) \wedge \delta_A, \rho_i \preceq k)]$, for all k in W.

Theorem 6 *The localized proof rule can be instantiated if and only if it can be instantiated with the strongest split invariant.*

This result (whose non-compositional analogue is also true) is useful for automatic calculation, as one can separate the calculation of the split invariant—which is done by the fixpoint procedure described previously—from the calculation of the split rank function.

12.2.8 Algorithms for Local Analysis of Temporal Properties

We suppose that a property is specified by a Büchi automaton, A, for its complement.

1. Compute the strongest split invariant, θ.
2. For each i, compute an abstraction, ${M_i}^{\theta}$, of component M_i relative to θ as follows. The initial states are those of M_i. The transition relation includes all transitions from T_i, and all interference transitions, $sum_j(\theta) \wedge unch(L_i)$, generated by every other component, j.
3. Form the synchronous compositions $C_i = {M_i}^{\theta} \times A$.
4. Use one of two tests for correctness, specified by Theorems 7 and 8 below.

Theorem 7 *The program satisfies the property if for* some *abstract component C_i, there is no infinite computation on which Green holds infinitely often.*

Proof Sketch Consider an erroneous global computation, σ. There is a run r of the automaton on σ in which a *Green* state occurs infinitely often. As θ is a split

invariant, it holds for all states on σ. Thus, each component of θ holds for every state of σ.

We project σ on to a component M_i to obtain a sequence δ, as follows. Consider each transition of σ. If it is a transition of T_i, it is retained as is; if not (say it is a transition by component M_j) it is replaced with the corresponding summary transition, $sum_j(\theta) \wedge unch(L_i)$. The replacement is possible as along the sequence, every state satisfies θ_j. The sequence δ is a computation of the abstract ${M_i}^{\theta}$ by construction. Since the shared state is preserved in δ, so is the accepting run of A; hence, δ induces an infinite computation of C_i where a *Green* state occurs infinitely often. □

Theorem 8 *The program satisfies the property if for* all *abstract components C_i, there is no infinite computation on which there are infinitely many T_i transitions from Green states.*

Proof Sketch This proof uses a refinement of the argument made in the proof sketch for Theorem 7. As a *Green* state occurs infinitely often in the erroneous computation σ, there is a component M_i such that infinitely many of the transitions originating at *Green* states are T_i transitions. Hence, the projection of σ on M_i, which is a computation of ${M_i}^{\theta}$, satisfies the stronger property that infinitely often there is a transition of T_i from a *Green* state. This fails the test in the statement of the theorem. □

Theorem 8 provides the more accurate of the two tests. For the mutual exclusion example, the property "infinitely often, `x=1`" can be proved using the test in Theorem 8, but not with the test in Theorem 7.

Local proof rules for fairness and the corresponding automated method are discussed in [25]. A fairness assumption typically refers to the transitions of all processes, an example is the unconditional fairness assumption: "every process transition is taken infinitely often". While it is possible to construct a Büchi automaton for properties under fairness, this automaton cannot be limited to the shared states. Thus, handling fairness compositionally requires a new approach. The key idea in [25] is to start with a weaker fairness assertion over the shared memory state and strengthen it iteratively as needed.

12.2.9 Automating the Discovery of Auxiliary Variables

As discussed in Sect. 12.2.5, the incompleteness that is inherent to the split invariance formulation can be remedied by adding auxiliary shared variables. Thus, a fully automated method also requires the development of automatic methods for discovering auxiliary variables. In this section, we describe a basic scheme which applies the CEGAR (CounterExample-Guided-Abstraction-Refinement) principle, in a manner specialized to compositional reasoning.

The overall method is iterative: at each iteration, a split invariant is computed. If the invariant does not suffice to prove the desired property (or to demonstrate failure), new auxiliary Boolean variables are added to the shared state, and the next iteration is initiated. Each Boolean variable corresponds to an assertion over the local state of some process. In effect, each variable exposes a part of the internal state of a component at the shared level. The result is that the split invariant which is computed at the next iteration is stronger than the current split invariant. In the limit, all of the local state is exposed as shared state, and the compositional calculation is then identical to a reachability analysis over the global program states. The best situation, however, is clearly that where exposing only a small subset of the local state suffices to compute a strong split invariant.

A heuristic algorithm for discovery of local assertions is given in [23]. If the strongest split invariant, θ, does not imply a global invariant φ, the heuristic looks for a distinguishing pair of states, say (s, t), such that both states satisfy θ and agree on the shared state (including all previously added auxiliary variables); however, s satisfies φ but t does not. If the pair differs in the local state for component i, a new auxiliary Boolean variable corresponding to the local predicate $(L_i = s(i))$ is added. Unlike non-compositional applications of CEGAR, the new predicate refers to the state of a single component. It is possible that no distinguishing pair can be found; in this case, the predecessors of the states in $(\theta \wedge \neg\varphi)$ are also marked as error states. A key property of this heuristic is that it is complete: eventually, either enough auxiliary predicates are added to obtain a valid proof, or a real error is discovered. A drawback is that the set of error states is not represented as a separable assertion.

For the mutual exclusion example from Sect. 12.2.5, this heuristic adds an auxiliary predicate for each component which indicates whether the component is in its critical state, C. Taken together, the auxiliary predicates are equivalent to the auxiliary variable *last*: at most one of them can be true, and that one indicates the last component to enter its critical section. Experiments in [24] show that the compositional calculation with the refinement step (implemented symbolically using BDDs) can be significantly faster, by an order of magnitude or more, than the global reachability computation. This gap can be larger for explicit state implementations, as the use of BDDs ameliorates state explosion to an extent.

For liveness properties, the problem of discovering auxiliary predicates is somewhat simpler as the correctness property is restricted to the shared state. The refinement method is based on the abstract components which are defined in Sect. 12.2.8. A violation gives rise to a counterexample trace in some abstract component. This trace can be enlarged to a global counterexample if all of the summary transitions along it are MUST-transitions. A summary transition from a state $(X = a, L_i = b)$ in ${M_i}^{\theta}$ that originates from process M_j is a MUST-transition if it is enabled at all states of the form $(X = a, L_j = c)$ which belong to θ_j. If it is not a MUST-transition, the set of local states $\{c\}$ where the MUST-condition fails forms an auxiliary predicate for M_j. It is shown in [24] that this strategy is complete: eventually, either enough auxiliary predicates are added to obtain a valid proof, or a real error is discovered.

12.2.10 Local Symmetry

It is often the case that concurrent programs exhibit non-trivial symmetries. Using symmetries on the global state space, one can reduce the number of states that must be analyzed for model checking [20, 32, 56]. For many programs, however, the global state space has little symmetry, and the reduction is correspondingly less effective. For example, protocols on ring or torus networks of N nodes typically have a symmetry group of size $O(N)$, giving a linear reduction of a potentially exponential state space. Distributed protocols may operate on irregular networks which have little to no symmetry. Compositional reasoning opens up the possibility, in such cases, of exploiting *local symmetries*, which are more prevalent. In a ring or torus network, for instance, any two nodes have isomorphic neighborhoods and are thus locally similar.

In a process network, a structural *local symmetry* is a triple of the form (m, β, n), where m and n are nodes, and β is an isomorphism between the network neighborhoods of m and n. The set of all local symmetries of a network forms a *groupoid* [43]. This is a set with group-like properties: for instance, if (m, β, n) is a local symmetry, the tuple (n, β^{-1}, m) is its inverse, and symmetries (m, β, n) and (n, γ, k) may be composed to form the symmetry $(m, \gamma\beta, k)$. Golubitsky and Stewart, in [43], define a recursive notion of local symmetry, called *balance*. This has a form akin to the co-inductive definition of bisimulation: roughly, for balanced nodes m and n, and any neighbor j of m, there is a neighbor k of n such that j and k are balanced. The significance of this formulation is shown by the following theorem, from [74]. For a symmetry (m, β, n), let $\beta(X)$ be the function which maps a set X of neighborhood states of node m to corresponding neighborhood states for node n, via the isomorphism β.

Theorem 9 *Let θ^* be the strongest split invariant on a network. If (m, β, n) is part of a balance relation on the network, then* $[\theta^*(n) \equiv \beta(\theta^*(m))]$.

This theorem shows that balanced nodes have isomorphic components in the strongest split invariant. Hence, it suffices to compute the split invariant for a single representative of each balance equivalence class. For uniform networks such as rings and tori, which have limited global symmetry, there is a *single* balance class, as any two nodes are balanced. Of course, networks such as star and complete networks, which have considerable global symmetry, also have a single balance class. Thus, in these networks, the cost of calculating a split invariant is reduced to computing a single component in place of all N. For the ring and torus networks, which have bounded degree, this cost is independent of the size of the network. For irregular networks, per-neighborhood abstraction can induce local symmetries where none originally exist, as is explored in [75].

12.2.11 Further Reading

The computation of a split invariant was originally formulated in explicit-state terms in [38]. Fixpoint computations for constructing separable predicates in the synchronous (hardware) model are described in [19, 70]. The chaotic iteration theorem makes it possible to parallelize the computation of a split invariant [26]. Alternative strategies for computing auxiliary variables using constraint solvers are developed in [47]. Connections between compositional proofs and sequentializing transformations of concurrent programs are explored in [39]. Local proof techniques focusing on data flow are developed in [34, 35]. There are close connections between local symmetry, compositional reasoning, and parametric proofs. A number of techniques [3, 72, 75, 81] transform compositional invariants of small instances of a parametric system to quantified invariants which hold for all instances.

12.3 Automata-Based Assume-Guarantee Reasoning

In this section we introduce assume-guarantee style reasoning for systems made up of components modeled as finite-state machines (FSMs). Properties and assumptions are also represented as finite-state machines. We provide the necessary background: we define finite-state machines and their parallel composition and present how safety properties are checked.

We then introduce assume-guarantee reasoning and the notion of the weakest assumption. We first present a simple assume-guarantee rule and consequently describe a framework for automating it. Subsequently, we discuss other assume-guarantee rules (asymmetric, symmetric, and circular) in the context of checking safety and liveness properties. Finally, we discuss the more general problem of interface generation, and close the chapter with a discussion of related work and further reading suggestions.

12.3.1 Formalisms

12.3.1.1 Finite-State Machines

Let $\mathcal{A}ct$ be the universal set of observable actions and let τ denote a local action *unobservable* to a component's environment.

An FSM M is a five-tuple $\langle Q, \alpha M, \delta, q_0, F \rangle$ where:

- Q is a finite non-empty set of states,
- $\alpha M \subseteq \mathcal{A}ct$ is a set of observable actions called the *alphabet* of M,
- $\delta \subseteq Q \times (\alpha M \cup \{\tau\}) \times Q$ is a transition relation,
- $q_0 \in Q$ is the initial state,
- $F \subseteq Q$ is a set of accepting states.

A transition $(s, a, s') \in \delta$ is written as $s \xrightarrow{a} s'$. A *trace* t of an FSM M is a finite sequence of observable actions that label the transitions that M can perform starting at its initial state (ignoring the τ-transitions). For an FSM M and a trace t, let $\hat{\delta}(q, t)$ denote the set of states that M can reach after reading t starting at state q. A trace t is said to be *accepted* by an FSM $M = \langle Q, \alpha M, \delta, q_0, F \rangle$ if $\hat{\delta}(q_0, t) \cap F \neq \emptyset$. The *language accepted by* M, denoted $\mathcal{L}(M)$, is the set $\{t \mid \hat{\delta}(q_0, t) \cap F \neq \emptyset\}$.

We sometimes denote by t both a trace and its trace FSM. For a trace t of length n, its trace FSM consists of $n + 1$ states, all accepting, where there is a transition between states m and $m + 1$ on the mth action in the trace t.

For $\Sigma \subseteq \mathcal{A}ct$, we use $t \uparrow \Sigma$ to denote the trace obtained by removing from t all occurrences of actions $a \notin \Sigma$. Similarly, $M \uparrow \Sigma$ is defined to be an FSM over alphabet Σ which is obtained from M by renaming to τ all the transitions labeled with actions that are not in Σ. Let t, t' be two traces. Let Σ, Σ' be the sets of actions occurring in t, t', respectively. By the *symmetric difference* of t and t' we mean the symmetric difference of the sets Σ and Σ'.

An FSM M is *non-deterministic* if it contains τ-transitions or if there exists $(q, a, q'), (q, a, q'') \in \delta$ such that $q' \neq q''$. Otherwise, M is *deterministic*.

12.3.1.2 Parallel Composition of FSMs

Let $M_1 = \langle Q^1, \alpha M_1, \delta^1, q_0^1, F^1 \rangle$ and $M_2 = \langle Q^2, \alpha M_2, \delta^2, q_0^2, F^2 \rangle$ be two FSMs. The parallel composition operator $\parallel$ is a commutative and associative operator that combines the behavior of two components by synchronizing the actions common to their alphabets and interleaving the remaining actions. Formally, $M_1 \parallel M_2$ is an FSM $M = \langle Q, \alpha M, \delta, q_0, F \rangle$, where $Q = Q^1 \times Q^2$, $q_0 = (q_0^1, q_0^2)$, $\alpha M = \alpha M_1 \cup \alpha M_2$, and δ is defined as follows, where $s_1, s_1' \in Q^1$ and $s_2, s_2' \in Q^2$ (note that the symmetric rules are implied by the fact that the operator is commutative):

$$\frac{s_1 \xrightarrow{a} s_1', a \notin \alpha M_2}{(s_1, s_2) \xrightarrow{a} (s_1', s_2)} \qquad \frac{s_2 \xrightarrow{a} s_2', a \notin \alpha M_1}{(s_1, s_2) \xrightarrow{a} (s_1, s_2')}$$

$$\frac{s_1 \xrightarrow{a} s_1', s_2 \xrightarrow{a} s_2', a \neq \tau}{(s_1, s_2) \xrightarrow{a} (s_1', s_2')}$$

The language of $M_1 \parallel M_2$ is $\mathcal{L}(M_1 \parallel M_2) = \{t \mid t \uparrow \alpha M_1 \in \mathcal{L}(M_1) \wedge t \uparrow \alpha M_2 \in \mathcal{L}(M_2) \wedge t \in (\alpha M_1 \cup \alpha M_2)^*\}$.

12.3.1.3 Properties

We will first introduce assume-guarantee reasoning in the context of checking safety properties. Later in this section we will also talk about liveness properties. For the context of our presentation, a safety property is modeled as an FSM P, whose language $\mathcal{L}(P)$ defines the set of acceptable behaviors over αP. For FSMs M and P where $\alpha P \subseteq \alpha M$, $M \models P$ if and only if

$$\forall t \in \mathcal{L}(M) : t \uparrow \alpha P \in \mathcal{L}(P)$$

12.3.1.4 Complementation

The complement of an FSM M, denoted coM, is an FSM that accepts the complement of M's language. It is constructed by first making M deterministic, subsequently completing it with respect to αM, and finally turning all accepting states into non-accepting ones, and vice-versa. An automaton is complete with respect to some alphabet if every state has an outgoing transition for each action in the alphabet. Completion typically introduces a non-accepting state and appropriate transitions to that state.

12.3.2 Assume-Guarantee Reasoning

12.3.2.1 Assume-Guarantee Triples

In the assume-guarantee paradigm a formula is a triple $\langle A \rangle\ M\ \langle P \rangle$, where M is a component, P is a property, and A is an assumption about M's environment. The formula is true if whenever M is part of a system satisfying A, then the system also guarantees P, i.e., $\forall E,\ E \parallel M \models A$ implies $E \parallel M \models P$. Assume-guarantee triples can be checked through reachability of error states in $A \parallel M \parallel coP$ [78]. A state (s_A, s_M, s_{coP}) is an error state if s_{coP} is accepting in coP.

12.3.2.2 Weakest Assumption

Let M be a finite-state component with Σ being the set of its interaction points with the environment, and let P be a safety property. Then there is a natural notion of the *weakest assumption* A_w, such that $\langle A_w \rangle\ M\ \langle P \rangle$ holds, where $\alpha A_w = \Sigma$. A_w characterizes all the possible environments E under which the property holds, i.e.,

$$\forall E : M \parallel E \models P \quad \text{iff} \quad E \models A_w.$$

It has been shown that, for any finite-state component M, the weakest assumption A_w exists, and can be constructed algorithmically [41]. The weakest assumption is associated with a notion of precision defined in the literature for "temporal" component interfaces [48], i.e., interfaces that capture ordering relationships between invocations of component methods. For example, an interface may describe the fact that closing a file before opening it is undesirable because an exception will be thrown. An ideal interface should precisely represent the component in all its intended usages. It should be *safe*, meaning that it should exclude all problematic interactions, and *permissive*, in that it should include all good interactions [48].

Safety and permissiveness can similarly be defined for assumptions, where the weakest assumption is the one that is both safe and permissive. An assumption A is safe if $\langle A \rangle\ M_1\ \langle P \rangle$. Safety is concerned with restricting behaviors to only those that satisfy P. Permissiveness is concerned with including behaviors, making sure that

behaviors are restricted only if necessary. Permissiveness is desirable because A_w is then appropriate for deciding whether an environment E is suitable for M_1 (if E does not satisfy A_w, then $E \parallel M_1$ does not satisfy P).

12.3.2.3 Basic Assume-Guarantee Rule

The simplest assume-guarantee rule is for checking a safety property P on a system with two components, M_1 and M_2.

Rule ASYM

$$\frac{1 : \langle A \rangle\ M_1\ \langle P \rangle \qquad 2 : \langle \mathit{true} \rangle\ M_2\ \langle A \rangle}{\langle \mathit{true} \rangle\ M_1 \parallel M_2\ \langle P \rangle}$$

In this rule, A denotes an assumption about the environment of M_1. Note that the rule is not symmetric in its use of the two components, and does not support circularity. Despite its simplicity, experience has shown it to be quite useful in the context of checking safety properties.

12.3.2.4 Soundness and Completeness

Soundness of an assume-guarantee rule means that whenever its premises hold, its conclusion holds as well. Without soundness, we cannot rely on the correctness of conclusions reached by applications of the rule, which makes the rule useless for verification. On the other hand, completeness states that whenever the conclusion of the rule is correct, the rule is applicable, i.e., there exist suitable assumptions such that the premises of the rule hold. While completeness is not needed to ensure correctness of proofs obtained by the rule, it is important as a measure for the usability of the rule. Rule ASYM is both sound and complete.

To prove soundness, we assume that the two premises hold and show that $\langle \mathit{true} \rangle\ M_1 \parallel M_2\ \langle P \rangle$ also holds. By definition of assume-guarantee triples, we need to show that for any environment E: $(E \parallel M_1 \parallel M_2) \models \mathit{true}$ implies $(E \parallel M_1 \parallel M_2) \models P$. Let E be an arbitrary environment such that $(E \parallel M_1 \parallel M_2) \models \mathit{true}$. We instantiate the definition of assume-guarantee triples for premise 1 and premise 2 with environments $(E \parallel M_2)$, and $(E \parallel M_1)$, respectively. We thus obtain that (1) $(E \parallel M_1 \parallel M_2) \models A$ implies $(E \parallel M_1 \parallel M_2) \models P$, and (2) $(E \parallel M_1 \parallel M_2) \models \mathit{true}$ implies $(E \parallel M_1 \parallel M_2) \models A$. From these two we conclude that $(E \parallel M_1 \parallel M_2) \models P$, as desired. Completeness holds trivially, by substituting M_2 for A. □

For the use of rule ASYM to be justified, the assumption should be (much) smaller than M_2, but still reflect M_2's behavior, i.e., A should be an abstraction of M_2, according to premise 2. Additionally, an appropriate assumption for the rule needs to "restrict" M_1 enough to satisfy P in premise 1. Coming up with such assumptions manually is highly non-trivial. In the next section we describe techniques for synthesizing assumptions automatically.

12.3.3 Automation of Basic Assume-Guarantee Rule

In this section, we focus on techniques for automating assume-guarantee reasoning using Rule ASYM. In particular, the framework that we describe uses the L* automata-learning algorithm to provide, and gradually refine, approximations of the desired assumption.

12.3.3.1 The L* algorithm

L* is a learning algorithm that was developed by Angluin [10] and later improved by Rivest and Schapire [82]. L* learns an unknown regular language and produces a deterministic finite-state machine (DFM) that accepts it. Let U be an unknown regular language over some alphabet Σ. In order to learn U, L* needs to interact with a *Minimally Adequate Teacher*, from now on called a *Teacher*. A Teacher must be able to correctly answer two types of questions from L*. The first type is a *membership query*, consisting of a string $\sigma \in \Sigma^*$; the answer is *true* if $\sigma \in U$, and *false* otherwise. The second type is an *equivalence query*, or *conjecture*, i.e., a candidate DFM C whose language the algorithm believes to be identical to U. The answer is *true* if $\mathcal{L}(C) = U$. Otherwise, the Teacher returns a counterexample, which is a string σ in the symmetric difference of $\mathcal{L}(C)$ and U.

L* creates a table where it incrementally records whether strings in Σ^* belong to U. It does this by making membership queries to the Teacher. At various stages L* decides to make a conjecture. It constructs a candidate automaton C based on the information contained in the table, and asks the Teacher whether the conjecture is correct. If it is, the algorithm terminates. Otherwise, L* uses the counterexample returned by the Teacher to extend the table with strings that witness differences between $\mathcal{L}(C)$ and U.

Characteristics of L.* L* depends on the correctness of the Teacher in order to provide a number of guarantees. More specifically, L* is guaranteed to terminate with a minimal automaton for the unknown language U. Moreover, each candidate DFM C that L* constructs is smallest, in the sense that any other DFM consistent with the information provided to L* has at least as many states as C. This characteristic of L* makes it particularly attractive in the context of learning interfaces or assumptions, since in the frameworks that we describe, the candidates provided by L* are combined with component models in model-checking steps. Smaller state machines typically result in easier model-checking problems. The conjectures made by L* strictly increase in size; each conjecture is smaller than the next one, and all incorrect conjectures are smaller than the minimal automaton for language U. Therefore, if that minimal automaton has n states, L* makes at most $n - 1$ incorrect conjectures. The number of membership queries made by L* is $\mathcal{O}(kn^2 + n \log m)$, where k is the size of the alphabet of U, n is the number of states in the minimal DFM for U, and m is the length of the longest counterexample returned when a conjecture is made.

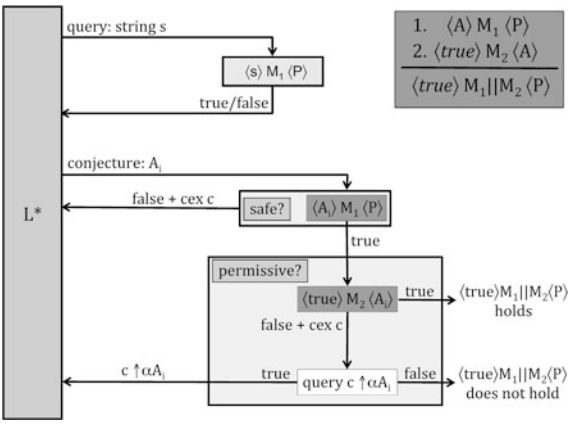


Fig. 2 Learning assumptions for assume-guarantee reasoning

12.3.3.2 Learning-Based Assumption Generation

From the definition of the weakest assumption A_w, one can observe that with A_w, the premises of Rule ASYM become necessary, in addition to being sufficient, for the conclusion of the rule to hold. In other words, $(\langle A_w \rangle\ M_1\ \langle P \rangle)$ and $(\langle true \rangle\ M_2\ \langle A_w \rangle)$ hold if and only if $(\langle true \rangle\ M_1 \parallel M_2\ \langle P \rangle)$. This is an advantage for an automated assume-guarantee reasoning framework, since it enables us to also disprove properties of a system, compositionally.

The framework illustrated in Fig. 2, and first presented in [22], provides a learning-based automation for assume-guarantee reasoning with Rule ASYM. In this framework, L* targets the computation of the weakest assumption A_w, for the reasons stated above. The set of interaction points of component M_1 with its environment, constituting the alphabet of the weakest assumption in this context, hence the alphabet over which L* is learning, is defined as: $(\alpha M_1 \cup \alpha P) \cap \alpha M_2$. Note that the framework uses the knowledge of the actual environment of component M_1, namely, component M_2, to make the reasoning more efficient. More specifically, the framework implements a teacher for L*, meaning that it responds to membership and equivalence queries, as described in the following.

Membership Queries. L* is first used to repeatedly *query* M_1 to check whether, in the context of strings s, M_1 violates the property. More formally, the query corresponds to checking the triple $\langle s \rangle\ M_1\ \langle P \rangle$ as illustrated in Fig. 2 (note that s represents its trace FSM). Checking $\langle s \rangle\ M_1\ \langle P \rangle$ corresponds to simulating string s on $M_1 \parallel P$: if an error is reachable, then the triple is *false*, otherwise it is *true*. The query returns *true/false* if $\langle s \rangle\ M_1\ \langle P \rangle$ is *true/false*, respectively. This is because, as mentioned, A_w allows all behaviors that satisfy the property, and disallows only violating behaviors.

Equivalence Queries. The conjectured automaton A is checked for correctness, which in this context means checking whether it corresponds to the weakest assumption or not. Note that, in Fig. 2, we use A_i to denote the ith assumption conjectured by the framework, which we simply refer to as A here. As discussed earlier, the weakest assumption is safe and permissive. We therefore reduce equivalence queries

to two separate checks, for safety and permissiveness of A, which we name Oracle 1 and Oracle 2, respectively.

Oracle 1 checks whether A is *safe*, by checking triple $\langle A \rangle\ M_1\ \langle P \rangle$, using a model checker. If A is safe, then the Teacher proceeds to Oracle 2. If it is unsafe, the model checker returns a counterexample t. The resulting counterexample t, projected on the assumption alphabet αA_w, is returned to L* to refine its conjecture. The projection is necessary because L* needs counterexamples in terms of the alphabet over which it is learning.

Oracle 2 checks whether safe assumption A is also permissive. As discussed, permissiveness is concerned with ensuring that the assumption does not exclude correct behaviors. However, given the fact that the main goal of the framework is to prove or disprove a property of the system using assume-guarantee reasoning, the framework does not need to generate a fully permissive assumption. Rather, it uses M_2 to add behaviors to over-restrictive assumptions on demand, and as needed for completion of the verification.

Note that Oracle 1, in essence, checks that premise 1 of Rule ASYM holds. To prove the property on the system using this rule, one would need to additionally check premise 2: ($\langle true \rangle\ M_2\ \langle A \rangle$). We therefore use premise 2 as follows, to drive the permissiveness check and to thereby potentially complete assume-guarantee reasoning (see Fig. 2).

If $\langle true \rangle\ M_2\ \langle A \rangle$, which consists of a model-checking step, is *true*, then we know that both premises of Rule ASYM hold, and therefore P holds for $M_1 \parallel M_2$. If $\langle true \rangle\ M_2\ \langle A \rangle$ is *false*, the Teacher performs some analysis to determine the underlying reason (see Fig. 2). This analysis consists of a simulation step identical to the one performed to respond to membership queries, as presented above. Specifically, the Teacher performs a query to determine whether the returned counterexample cex, projected on the alphabet of the assumption, belongs to A_w, in which case L* needs to refine the assumption. If the query returns *true*, then A is not permissive, so $cex \uparrow \alpha A$ is returned to L* for refinement of its guess. If, on the other hand, the answer is false, it means that cex is a word that belongs to M_2, in the context of which M_1 violates the property P. As a consequence, $M_1 \parallel M_2$ does not satisfy the property P.

Notice that the answers that the framework provides to L* are always precise with respect to the targeted weakest assumption. However, the framework uses M_2 to select which missing words to include in the language of the assumption. The reason is that we restrict our reasoning to a specific context, rather than accounting for all possible contexts, as required for the computation of A_w. That means, of course, that the assumption obtained from this framework does not necessarily correspond to A_w. On the other hand, we remind the reader that the primary goal is to obtain conclusive results from the assume-guarantee rule. As soon as we are able to prove or disprove the property in the system, we stop refining the learned assumption, since we have achieved our goal. The assumption computed with this framework will be smaller than, or in the worst case equal to A_w, in terms of number of states, as guaranteed by the characteristics of L*. In the worst case, where A_w itself is computed, the framework is guaranteed to terminate, because A_w is

both necessary and sufficient, and therefore the framework will prove or disprove the property during this iteration.

12.3.3.3 Correctness Arguments

Framework correctness argument: The framework directly uses the assume-guarantee rule Rule ASYM to answer conjectures. Correctness of the rule guarantees correctness of positive answers by the framework. On the other hand, each counterexample reported is a real counterexample, as discussed above.

Teacher Correctness Argument: Correctness of the teacher corresponds to showing that all the answers returned to L* are consistent with A_w. This was discussed during the presentation of the framework above.

Termination Argument: Given the fact that our Teacher only comes back to L* for refinement with counterexamples related to A_w, the framework eventually converges to A_w, unless it terminates earlier. As discussed, A_w makes Rule ASYM sound and complete, and therefore our framework will return a conclusive answer at that iteration. As a result, the framework always terminates.

12.3.4 Example

Let us revisit the mutual exclusion protocol presented in Sect. 12.2.5. We model the protocol in a behavior-based fashion. In particular, we use the FSP input language of the LTSA tool [64] to model FSMs. FSP stands for "Finite State Process" and is a process algebra-like language (see [21], Chap. 32 of this Handbook). Detailed syntax and semantics are provided in [64]. In the example, "->" stands for action prefix, "|" stands for choice, and "||" stands for parallel composition. Indexing "[]" is used to parameterize specifications. Note that in the LTSA-generated figures, an indexed transition corresponds to multiple transitions, one for each index in the specified range.

```
// State-based description
var x: boolean; initially x=true

M1:                              M2:
  while (true) {                  while (true) {
  T: skip;                        T: skip;
  H: <if x then x:= false>        H: <if x then x:= false>
  E: x := true                    E: x := true
  }                               }


// Behavior-based implementation
// FSP code for LTSA model-checking tool

const False = 0
const True = 1
range Boolean = False..True

// variable implements atomic test-and-set
```

```
X = Var[True],
Var[True] = (m[1..2].atomicTestSetX[False] -> Var[False]),
Var[False] = (m[1..2].setX[True] -> Var[True]).


// module implementation
M(Steps=1) = Hungry,
Hungry = (atomicTestSetX[False] -> Eating),
Eating = (start_eating -> CS[0]),
CS[i:0..Steps] = ( when (i< Steps) step[i]  -> CS[i+1]
                 | when (i==Steps) done_eating -> setX[True] -> M).


|| M1 = (m[1]:M(4)).  // creating module M1
|| M2 = (m[2]:M(5)).  // creating module M2


// mutual exclusion property
property
MX = (m[i:1..2].start_eating -> m[i].done_eating -> MX).

|| Module1 = ( X || M1 || MX ).

set Alphabet0 = {m[2].{setX[True], atomicTestSetX[False],
                start_eating, done_eating}}

|| Module2 = (M2).
```

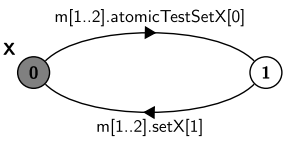
We model the two concurrent modules $M1$ and $M2$ by creating two instances of the FSM M named $m[1]$ and $m[2]$, respectively. The FSM M is parameterized by the number of steps that each module performs in its "eating" stage. For example, $M1$ has four processing steps, and $M2$ has five. This models the fact that a module may include several states within its critical section. Figure 4 illustrates $M1$ for a single processing step. All figures in this section have been created automatically by the LTSA tool.

Variable X allows the two modules to atomically set it to *False* when its current value is *True*, and allows them to set its current value to *True* unconditionally. The FSM corresponding to this variable implementation is illustrated in Fig. 3. The mutual exclusion property requires that $M1$ and $M2$ not be eating at the same time: each module must finish eating before the other module starts eating. This is illustrated in Fig. 5. In all the FSMs presented in this section state "0" is initial and all the states are accepting.

For compositional verification, we group $M1$ with variable X and the property MX into *Module*1, and $M2$ into *Module*2. The alphabet of the interface between the two modules is defined as *Alphabet*0. The assumption generated by the learning framework is depicted in Fig. 6.

The generated assumption has four states. The module that it represents, *Module*2, has five states for one processing step, and nine for four processing steps. Note that, no matter how many processing steps we include in the two modules,

Fig. 3 Shared variable



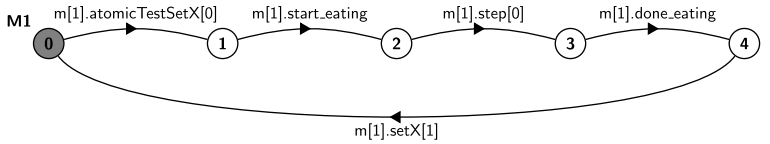


Fig. 4 Module M1 FSM for one processing step

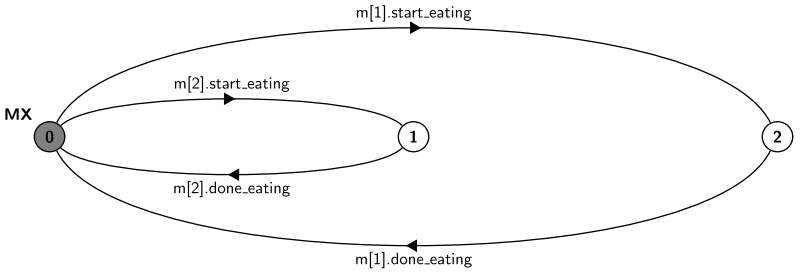


Fig. 5 Mutual exclusion property

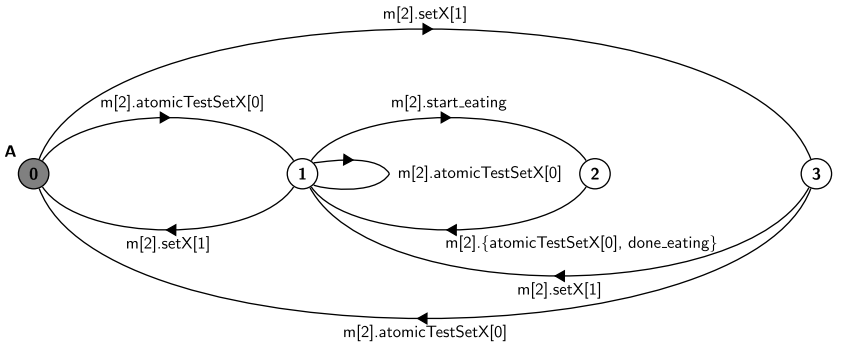


Fig. 6 Generated assumption

the assumption that is generated will always be exactly the same. The gains from compositional verification therefore become more pronounced for components that include many processing steps in their critical section. This confirms the fact that the algorithms presented are able to detect the main synchronization principle on which the correctness of this protocol is based, irrespective of the internal details of each module. More generally, generated assumptions may be much smaller than the system components, as they essentially abstract away the internal component computations and only represent (succinctly) the component interactions.

12.3.5 Additional Assume-Guarantee Rules and Their Automation

We note that rule ASYM can be extended to reason about more than two components. To see this, consider checking whether system $M_1 \parallel M_2 \parallel \cdots \parallel M_n$ satisfies P. One can decompose the system into: M_1 and $M_2' = M_2 \parallel M_3 \parallel \cdots \parallel M_n$ and apply the rule to reason about the two components M_1 and M_2'. Now notice that when checking the second premise of the rule, i.e., when checking whether triple $\langle true \rangle\ M_2'\ \langle A \rangle$ holds, assumption A plays the role of a safety property for system M_2'. Therefore, we can recursively apply rule ASYM when reasoning about $M_2' = M_2 \parallel \cdots M_n$, by decomposing M_2' into M_2 and $M_3' = M_3 \parallel \cdots M_n$ and so on. To summarize, we have the following rule for reasoning about $n \geq 2$ components [78]:

Rule ASYMN

$$
\begin{array}{ll}
1: & \langle A_1 \rangle\ M_1\ \langle P \rangle \\
2: & \langle A_2 \rangle\ M_2\ \langle A_1 \rangle \\
3: & \langle A_3 \rangle\ M_3\ \langle A_2 \rangle \\
 & \cdots \\
n-1: & \langle A_{n-1} \rangle\ M_{n-1}\ \langle A_{n-2} \rangle \\
n: & \langle true \rangle\ M_n\ \langle A_{n-1} \rangle \\
\hline
 & \langle true \rangle\ M_1 \parallel M_2 \parallel \cdots M_n\ \langle P \rangle
\end{array}
$$

Reasoning based on this rule can be similarly automated using learning techniques for the assumption inference, as explained below. To check whether system $M_1 \parallel M_2 \parallel \cdots M_n$ satisfies property P, consider the decomposition: M_1 and $M_2' = M_2 \parallel M_3 \parallel \cdots M_n$. The learning framework for two components described above can be applied recursively to check the second premise of the assume-guarantee rule, involving M_2', where the assumption A_1 for M_1 plays the role of property when checking M_2'. More generally, at each recursive invocation for M_j and $M_{j+1}' = M_{j+1} \parallel M_{j+2} \parallel \cdots M_n$ we use learning to infer an assumption A_j such that both $\langle A_j \rangle\ M_j\ \langle A_{j-1} \rangle$ and $\langle true \rangle\ M_{j+1} \parallel M_{j+2} \parallel \cdots M_n\ \langle A_j \rangle$ hold, and the second triple is checked through another invocation of the learning framework.

Although sound and complete, the two rules presented so far are not always satisfactory since they are not symmetric in the use of the components. Some form of circular or symmetric rules is desirable, that makes use of all the components in a similar way.

The obvious "circular" rule for the parallel composition of two components discharges the two assumption of each component by the guarantee of the other component:

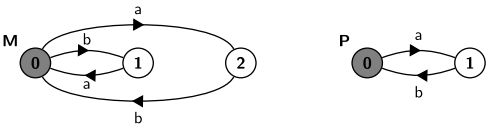


Fig. 7 Example that illustrates unsoundness of rule CIRC-0

Rule CIRC-0

$$\frac{\begin{array}{l}1 : \langle A_1 \rangle\ M_1\ \langle P \rangle \\ 2 : \langle A_2 \rangle\ M_2\ \langle P \rangle \\ 3 : P \models A_1 \wedge A_2\end{array}}{M_1 \parallel M_2 \models P}$$

However, this rule is not sound as demonstrated by the example in Fig. 7. Consider components M_1 and M_2 and property P; both M_1 and M_2 have the same behavior as M (see figure). Now consider two assumptions A_1 and A_2 that have the behavior defined by P. Then premise 3 holds. Premises 1 and 2 also hold, since A_1 restricts M_1 to be exactly P, similarly for premise 2. Therefore, according to rule CIRC-0, $M_1 \parallel M_2$ satisfies P. On the other hand $M_1 \parallel M_2$ is M again which violates P, since it allows b to occur before a. The circular reasoning involved in this rule is unsound. The rule fails essentially because the two components may have common behavior that violates the property, but this behavior is ruled out by the assumptions and it is never checked in the premises.

Circularity in itself is not a reason for unsoundness: other models of computation have sound circular rules [6, 59]. The soundness argument there relies on induction over finite traces and properties of the models. This is discussed later in the section.

In our setting, there are simple ways to remedy unsoundness. One way is to avoid circularity. Rule SYM below is an example of a rule that is *symmetric* in its use of components but does not use circular reasoning. Using this rule for automated compositional verification may result in smaller verification problems (than using rule ASYM) [78].

Rule SYM

$$\frac{\begin{array}{l}1 : \langle A_1 \rangle\ M_1\ \langle P \rangle \\ 2 : \langle A_2 \rangle\ M_2\ \langle P \rangle \\ 3 : \mathcal{L}(coA_1 \parallel coA_2) \subseteq \mathcal{L}(P)\end{array}}{M_1 \parallel M_2 \models P}$$

We require $\alpha P \subseteq \alpha M_1 \cup \alpha M_2 \cup \cdots \cup \alpha M_n$ and that for $i \in \{1, 2, \ldots n\}$

$$\alpha A_i \subseteq (\alpha M_1 \cap \alpha M_2 \cap \cdots \cap \alpha M_n) \cup \alpha P.$$

Informally, each A_i is a postulated environment assumption for the component M_i to achieve the safety property P. Recall that coA_i is the complement of A_i. Premise 3 of the rule ensures that all the possible common behaviors of M_1 and M_2 that are ruled out by both assumptions A_1 and A_2 satisfy property P (see the cause of unsoundness in the example above).

Rule SYM extends naturally to $n > 2$ components:

Rule SYMN

$$\frac{\begin{array}{ll} 1: & \langle A_1 \rangle\ M_1\ \langle P \rangle \\ 2: & \langle A_2 \rangle\ M_2\ \langle P \rangle \\ & \cdots \\ n: & \langle A_n \rangle\ M_n\ \langle P \rangle \\ n+1: & \mathcal{L}(coA_1 \parallel coA_2 \parallel \cdots coA_n) \subseteq \mathcal{L}(P) \end{array}}{M_1 \parallel M_2 \parallel \cdots M_n \models P}$$

Although sound and complete, the symmetric rules described above have two disadvantages: (i) they use the complement of the assumptions, which may be expensive to compute for non-deterministic assumptions, and (ii) the last discharge step uses the composition of *all* coA's, which may be expensive when n is large.

Alternative sound compositional proofs use circular reasoning principles in which properties of other components are assumed when proving properties of individual components. One such rule is the following:

Rule CIRC

$$\frac{\begin{array}{l} 1: \langle A_1 \rangle\ M_1\ \langle P \rangle \\ 2: \langle A_2 \rangle\ M_2\ \langle A_1 \rangle \\ 3: \langle true \rangle\ M_1\ \langle A_2 \rangle \end{array}}{M_1 \parallel M_2 \models P}$$

This rule is a special case of Rule ASYMN (for $n = 3$), where the first and last components coincide. The rule is sound and complete and it naturally extends to reasoning about systems containing more than two components. Both the symmetric and the circular rules discussed so far can be similarly automated using learning techniques for assumption generation. The interested reader should look in [78] for details.

Other models of composition have sound circular rules. These models have somewhat different notions of composition and conformance than the ones defined here. For instance, the following rule is sound for a special kind of FSMs called Moore machines [59] and in the more general setting of "Reactive Modules" [6]. The soundness proof is based on induction over the length of finite traces. It relies also on a crucial property of the models that allows every finite trace to be extended (cf. [6]).

Rule CIRC-IND

$$\frac{\begin{array}{l} 1: M_1 || P_2 \models P_1 \\ 2: M_2 || P_1 \models P_2 \end{array}}{M_1 || M_2 \models P_1 || P_2}$$

The inductive argument applies only to safety properties. To achieve soundness in general, the second hypothesis is strengthened to $M_2||\mathsf{safe}(P_1) \models P_2$, where $\mathsf{safe}(P_1)$ is the strongest safety property that includes the language of P_1 [6, 9]. The same rule is used in, e.g., [49, 50, 52] and it forms the basis of a systematic proof decomposition methodology [51]. Other circular rules have been proposed in e.g. [1, 67, 69, 80] and their soundness and completeness has been studied in [65, 73]. However, they have not been studied for automation.

12.4 Related Approaches

12.4.1 Learning for Compositional Verification

We have shown how to guide our learning for compositional verification towards an assumption that is both safe and permissive. Other researchers focused on the more computationally expensive problem of learning a *minimal* assumption [17, 46] for compositional verification, in other words, computing an assumption A_{min} such that any other assumption A that can check satisfaction or violation of P will have a greater or equal number of states, i.e., $|A| \geq |A_{min}|$.

So far, in our presentation, we have assumed that the alphabet of the assumptions learned for compositional verification is fixed as $(\alpha M_1 \cup \alpha P) \cap \alpha M_2$. However, it is sometimes possible to verify a problem with a smaller alphabet, resulting in potentially smaller assumptions and cheaper verification [15, 78].

Learning assumptions has also been applied in the context of symbolic and implicit model checking [16, 71], has been adapted for infinite traces in the context of assume-guarantee reasoning for liveness properties [33], and has been studied in the context of timed [63] and probabilistic systems [36, 58].

12.4.2 Assumption Generation by Abstraction-Refinement

The learning algorithm presented in Sect. 12.3.3.2 for assumption discovery basically starts from a small automaton and uses the off-the-shelf L* algorithm to split states based on queries that it makes and the counterexamples it receives. This is reminiscent of the well-known abstraction-refinement scheme (CEGAR), where some abstract description of a model is analyzed and iteratively refined based on spurious counterexamples that result from the abstraction being too coarse. Typically abstraction is designed to preserve correctness in some way (e.g., it may be an *over-approximation* of the original model). However, candidates produced by L* in the context of rule ASYM do not have clear semantic guarantees (i.e., being under- or over-approximations, not even when compared to each other). The focus of L* is to generate assumptions with the smallest number of states for the data it

gathers. An alternative approach [13] generates assumptions for the rule ASYM using *assume-guarantee abstraction-refinement* (AGAR), a variant of the well known CEGAR approach adapted to compositional reasoning (see [29], Chap. 13 of this Handbook). In this case, M_2 is abstracted in a conservative way, such that premise 2 holds by construction. However it is possible that premise 1 does not hold, and the counterexample returned is analyzed to see whether it corresponds to a real error or is spurious, due to the imprecision introduced by the abstraction. If the counterexample is spurious, the abstraction of M_2 is refined to eliminate it.

12.4.3 Components and Interfaces

At the heart of any modular development or reasoning technique is the capability to summarize aspects of a component that are relevant to its customers. These aspects are captured in component interfaces, which are closely related to the notion of assumptions.

As mentioned in Sect. 12.3.2, temporal interfaces capture ordering relationships between invocations of component methods. In such a context, component interfaces have the same flavor as assumptions that relate to safety properties, as studied in the previous section. The fundamental difference between an interface and an assumption in the context of compositional reasoning is that an interface summarizes the component *irrespective* of the environment in which the component is to be introduced. On the other hand, an assumption serves as a potentially imprecise interface that is sufficient for breaking up a targeted verification problem into simpler problems; all components that participate in the verification problem are known and available.

A precise interface is similar to the notion of a weakest component assumption. It is therefore characterized in terms of two properties, safety, and permissiveness. Note that error states in this context are not introduced by explicit properties, but are rather assumed to represent undesirable component states.

Interfaces can be learned through a very similar framework to that of Fig. 2. Queries and conjectures related to safety are answered in an identical way, except that in the case of interfaces we assume that error states exist in the components, whereas in the previous framework they come from the property P.

Permissiveness, however, needs to be different, because the environment of the targeted component is not available.

The example in Fig. 8 shows how a permissiveness check could be performed. Component M has states M_0, M_1, M_2, and M_{error} and states A_0, A_1, and A_{error} belong to an interface A. A permissiveness check needs to detect sequences that are blocked by the interface but are legal in the component. Such sequences identify that the interface is not permissive. This can be performed by checking reachability, in $M \parallel A$, of legal states of M and M_{error}. According to this check, trace a, b leading to state $[M_1, A_{error}]$ in the composition could be an indication that A is not permissive enough. However this is not true, since the same path leads to $[M_{error}, A_{error}]$. This

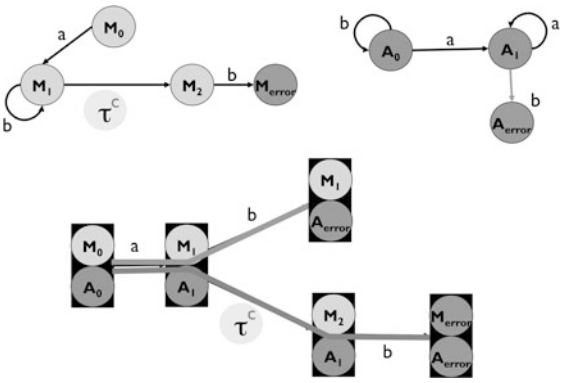


Fig. 8 Checking for permissiveness

happens because the alphabet of the assumption is $\{a, b\}$, meaning that action c in M is considered as a τ from the point of view of A. In the figure, this is illustrated as a τ action covering action c.

This example illustrates the fact that non-determinism in component M may cause spurious counterexamples in the permissiveness reachability check described above. As a consequence, precise characterization of permissiveness requires determinization of component M, which can be performed using a subset construction. The permissiveness check is therefore NP-hard [5], and can be inefficient in practice.

Several approaches have been proposed to deal with this problem. Unless determinization is a viable solution for a targeted component [12, 41], heuristic approaches are often used to determine whether a counterexample is spurious [5, 40]. Also, if non-determinism is introduced through abstraction of a deterministic concrete component, this problem can sometimes be avoided, using a combination of over- and under-approximating abstractions [85]. More recently, automata learning has been combined with symbolic execution or machine-learning approaches for the generation of interfaces enriched with method guards that represent constraints on method parameters for safe execution [42, 54, 87].

12.4.4 Other Modular Reasoning Frameworks

Assume-guarantee-style reasoning was introduced to help build and verify complex systems, where some form of component assumptions (either implicit or explicit) is needed. The assumptions are used to model the components' interaction with the environment (e.g., the other components) [57, 60].

Assume-guarantee reasoning has been studied extensively [6, 50, 52, 67] and large case studies are presented, for example, in [49, 66]. Some automated techniques to support the reasoning are presented in [7, 8, 37, 45]. However, these techniques still require some manual effort for the creation of the necessary assumptions. For example, the technique from [45] describes a framework for the assume-guarantee-style verification of properties written in the universal fragment of the

CTL temporal logic (ACTL) (see [79], Chap. 2 of this Handbook). A tableau method for ATCL, presented there, plays the role of an assumption. Also related is the work of Inverardi and colleagues, see [55] for example, which is also aimed at providing support for the modular verification of properties of interest, such as deadlock freedom. The Alternating Time Temporal Logic ATL (and transition systems) [7] was proposed for the specification and verification of open systems together with automated support via symbolic model-checking procedures. The Mocha toolkit [8] provides support for modular verification of components with requirement specifications based on the ATL.

The soundness of circular rules is justified essentially based on induction, although the form of the induction is different in each model of computation. A simplification and unification of such proofs is carried out in [2, 9].

Interface automata [4] provide an elegant formalism for the specification and verification of composite systems. Interface automata capture in the same model both input assumptions about the order in which the operations of a component are called and output guarantees about the order in which the component invokes operations on external components. The formalism supports automatic compatibility and refinement checks between interface models. These notions have game-theoretic foundations supported by efficient checking algorithms.

The underlying approach to automated assumption generation that we have presented here has similarities with "sub-module construction", "equation solving", "scheduler synthesis" and "supervisory control", as seen, for example, in [11, 62, 68] and in many works that followed. However, the goals of these methods are different than in compositional verification. There are many other approaches to compositional verification that are not based on assume-guarantee reasoning. An example related to the techniques presented here is compositional reachability analysis [18, 44], which is based on iterative composition and minimization with respect to properties of interest.

12.5 Conclusion

In this chapter we have discussed compositional reasoning techniques that address the fundamental challenge in model checking, namely the state explosion problem. We have focused on assume-guarantee style reasoning with particular emphasis on *automated* techniques for assumption generation, which we believe is essential for transitioning the techniques to practice. The subject of this chapter continues to be a very active area of research, see e.g. [14].

References

1. Abadi, M., Lamport, L.: Composing specifications. Trans. Program. Lang. Syst. **15**(1), 73–132 (1993)

2. Abadi, M., Merz, S.: An abstract account of composition. In: Wiedermann, J., Hájek, P. (eds.) Intl. Symp. on Mathematical Foundations of Computer Science (MFCS). LNCS, vol. 969, pp. 499–508. Springer, Heidelberg (1995)
3. Abdulla, P.A., Haziza, F., Holík, L.: All for the price of few. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) Intl. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI). LNCS, vol. 7737, pp. 476–495. Springer, Heidelberg (2013)
4. de Alfaro, L., Henzinger, T.A.: Interface automata. In: Intl. Symp. on Foundations of Software Engineering (FSE), pp. 109–120. ACM, New York (2001)
5. Alur, R., Cerný, P., Madhusudan, P., Nam, W.: Synthesis of interface specifications for Java classes. In: Palsberg, J., Abadi, M. (eds.) Symp. on Principles of Programming Languages (POPL), pp. 98–109. ACM, New York (2005)
6. Alur, R., Henzinger, T.A.: Reactive modules. Form. Methods Syst. Des. **15**(1), 7–48 (1999)
7. Alur, R., Henzinger, T.A., Kupferman, O.: Alternating-time temporal logic. J. ACM **49**(5), 672–713 (2002)
8. Alur, R., Henzinger, T.A., Mang, F.Y.C., Qadeer, S., Rajamani, S.K., Tasiran, S.: Mocha: modularity in model checking. In: Hu, A.J., Vardi, M.Y. (eds.) Intl. Conf. on Computer-Aided Verification (CAV), pp. 521–525 (1998)
9. Amla, N., Emerson, E.A., Namjoshi, K.S., Trefler, R.J.: Abstract patterns of compositional reasoning. In: Amadio, R.M., Lugiez, D. (eds.) Intl. Conf. on Concurrency Theory (CONCUR). LNCS, vol. 2761, pp. 423–438. Springer, Heidelberg (2003)
10. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. **75**(2), 87–106 (1987)
11. Aziz, A., Balarin, F., Brayton, R., DiBenedetto, M., Saldanha, A., Sangiovanni-Vincentelli, A.: Supervisory control of finite state machines. In: Wolper, P. (ed.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 939, pp. 279–292. Springer, Heidelberg (1995)
12. Beyer, D., Henzinger, T.A., Singh, V.: Algorithms for interface synthesis. In: Damm and Hermanns [28], pp. 4–19
13. Bobaru, M.G., Pasareanu, C.S., Giannakopoulou, D.: Automated assume-guarantee reasoning by abstraction refinement. In: Gupta, A., Malik, S. (eds.) Intl. Conf. on Computer-Aided Verification (CAV), vol. 5123, pp. 135–148. Springer, Heidelberg (2008)
14. Brookes, S.: A semantics for concurrent separation logic. Theor. Comput. Sci. **375**(1–3), 227–270 (2007). doi:10.1016/j.tcs.2006.12.034
15. Chaki, S., Strichman, O.: Three optimizations for assume-guarantee reasoning with L*. Form. Methods Syst. Des. **32**(3), 267–284 (2008)
16. Chen, Y.F., Clarke, E.M., Farzan, A., Tsai, M.H., Tsay, Y.K., Wang, B.Y.: Automated assume-guarantee reasoning through implicit learning. In: Touili, T., Cook, B., Jackson, P. (eds.) Intl. Conf. on Computer-Aided Verification (CAV), vol. 6174, pp. 511–526. Springer, Heidelberg (2010)
17. Chen, Y.F., Farzan, A., Clarke, E.M., Tsay, Y.K., Wang, B.Y.: Learning minimal separating DFAs for compositional verification. In: Kowalewski, S., Philippou, A. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 5505, pp. 31–45. Springer, Heidelberg (2009)
18. Cheung, S.C., Kramer, J.: Compositional reachability analysis of finite-state distributed systems with user-specified constraints. SIGSOFT Softw. Eng. Notes **20**(4), 140–150 (1995)
19. Cho, H., Hachtel, G.D., Macii, E., Plessier, B., Somenzi, F.: Algorithms for approximate FSM traversal based on state space decomposition. Trans. Comput.-Aided Des. Integr. Circuits Syst. **15**(12), 1465–1478 (1996)
20. Clarke, E.M., Filkorn, T., Jha, S.: Exploiting symmetry in temporal logic model checking. In: Courcoubetis, C. (ed.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 697, pp. 450–462. Springer, Heidelberg (1993)
21. Cleaveland, R., Roscoe, A., Smolka, S.A.: Process algebra and model checking. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)

22. Cobleigh, J.M., Giannakopoulou, D., Pasareanu, C.S.: Learning assumptions for compositional verification. In: Garavel, H., Hatcliff, J. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 2619, pp. 331–346. Springer, Heidelberg (2003)
23. Cohen, A., Namjoshi, K.S.: Local proofs for global safety properties. In: Damm and Hermanns [28], pp. 55–67. Full version in Formal Methods in System Design **34**(2) (2009)
24. Cohen, A., Namjoshi, K.S.: Local proofs for linear-time properties of concurrent programs. In: Gupta, A., Malik, S. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 5123, pp. 149–161. Springer, Heidelberg (2008)
25. Cohen, A., Namjoshi, K.S., Sa'ar, Y.: A dash of fairness for compositional reasoning. In: Touili, T., Cook, B., Jackson, P. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 6174, pp. 543–557. Springer, Heidelberg (2010)
26. Cohen, A., Namjoshi, K.S., Sa'ar, Y., Zuck, L.D., Kisyova, K.I.: Parallelizing a symbolic compositional model-checking algorithm. In: Barner, S., Harris, I.G., Kroening, D., Raz, O. (eds.) Intl. Haifa Verification Conference (HVC). LNCS, vol. 6504, pp. 46–59. Springer, Heidelberg (2010)
27. Cousot, P., Cousot, R.: Invariance proof methods and analysis techniques for parallel programs. In: Biermann, A., Guiho, G., Kodratoff, Y. (eds.) Automatic Program Construction Techniques, pp. 243–271. Macmillan, New York (1984). Chap. 12
28. Damm, W., Hermanns, H. (eds.): Computer Aided Verification, Proceedings of the 19th International Conference, CAV 2007, Berlin, Germany, July 3–7, 2007. LNCS, vol. 4590. Springer, Heidelberg (2007)
29. Dams, D., Grumberg, O.: Abstraction and abstraction refinement. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
30. Dijkstra, E., Scholten, C.: Predicate Calculus and Program Semantics. Springer, Heidelberg (1990)
31. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM **18**(8), 453–457 (1975)
32. Emerson, E.A., Sistla, A.P.: Symmetry and model checking. In: Courcoubetis, C. (ed.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 697, pp. 463–478. Springer, Heidelberg (1993)
33. Farzan, A., Chen, Y.F., Clarke, E.M., Tsay, Y.K., Wang, B.Y.: Extending automated compositional verification to the full class of omega-regular languages. In: Ramakrishnan, C.R., Rehof, J. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 4963, pp. 2–17. Springer, Heidelberg (2008)
34. Farzan, A., Kincaid, Z.: Verification of parameterized concurrent programs by modular reasoning about data and control. In: Field, J., Hicks, M. (eds.) Symp. on Principles of Programming Languages (POPL), pp. 297–308. ACM, New York (2012)
35. Farzan, A., Kincaid, Z., Podelski, A.: Inductive data flow graphs. In: Giacobazzi, R., Cousot, R. (eds.) Symp. on Principles of Programming Languages (POPL), pp. 129–142. ACM, New York (2013)
36. Feng, L., Han, T., Kwiatkowska, M.Z., Parker, D.: Learning-based compositional verification for synchronous probabilistic systems. In: Bultan, T., Hsiung, P. (eds.) Intl. Symp. Automated Technology for Verification and Analysis (ATVA). LNCS, vol. 6996, pp. 511–521. Springer, Heidelberg (2011)
37. Flanagan, C., Freund, S.N., Qadeer, S., Seshia, S.A.: Modular verification of multithreaded programs. Theor. Comput. Sci. **338**(1–3), 153–183 (2005)
38. Flanagan, C., Qadeer, S.: Thread-modular model checking. In: Ball, T., Rajamani, S.K. (eds.) Intl. Symposium on Model Checking of Software (SPIN). LNCS, vol. 2648, pp. 213–224. Springer, Heidelberg (2003)
39. Garg, P., Madhusudan, P.: Compositionality entails sequentializability. In: Abdulla, P.A., Leino, K.R.M. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 6605, pp. 26–40. Springer, Heidelberg (2011)

40. Giannakopoulou, D., Pasareanu, C.S.: Interface generation and compositional verification in JavaPathfinder. In: Chechik, M., Wirsing, M. (eds.) Intl. Conf. Fundamental Approaches to Software Engineering (FASE). LNCS, vol. 5503, pp. 94–108. Springer, Heidelberg (2009)
41. Giannakopoulou, D., Pasareanu, C.S., Barringer, H.: Component verification with automatically generated assumptions. Autom. Softw. Eng. **12**(3), 297–320 (2005)
42. Giannakopoulou, D., Rakamaric, Z., Raman, V.: Symbolic learning of component interfaces. In: Miné, A., Schmidt, D. (eds.) Intl. Symp. on Static Analysis (SAS). LNCS, vol. 7460, pp. 248–264. Springer, Heidelberg (2012)
43. Golubitsky, M., Stewart, I.: Nonlinear dynamics of networks: the groupoid formalism. Bull. Am. Math. Soc. **43**, 305–364 (2006)
44. Graf, S., Steffen, B.: Compositional minimization of finite state systems. In: Clarke, E.M., Kurshan, R.P. (eds.) Intl. Conf. on Computer-Aided Verification (CAV), vol. 531, pp. 186–196. Springer, Heidelberg (1990)
45. Grumberg, O., Long, D.E.: Model checking and modular verification. Trans. Program. Lang. Syst. **16**(3), 843–871 (1994)
46. Gupta, A., McMillan, K.L., Fu, Z.: Automated assumption generation for compositional verification. Form. Methods Syst. Des. **32**(3), 285–301 (2008)
47. Gupta, A., Popeea, C., Rybalchenko, A.: Predicate abstraction and refinement for verifying multi-threaded programs. In: Ball, T., Sagiv, M. (eds.) Symp. on Principles of Programming Languages (POPL), pp. 331–344. ACM, New York (2011)
48. Henzinger, T.A., Jhala, R., Majumdar, R.: Permissive interfaces. In: Wermelinger, M., Gall, H.C. (eds.) Intl. Symp. on Foundations of Software Engineering (FSE), pp. 31–40. ACM, New York (2005)
49. Henzinger, T.A., Liu, X., Qadeer, S., Rajamani, S.K.: Formal specification and verification of a dataflow processor array. In: White and Sentovich [86], pp. 494–499
50. Henzinger, T.A., Qadeer, S., Rajamani, S.K.: You assume, we guarantee: methodology and case studies. In: Hu, A.J., Vardi, M.Y. (eds.) Intl. Conf. on Computer-Aided Verification (CAV), vol. 1427, pp. 440–451. Springer, Heidelberg (1998)
51. Henzinger, T.A., Qadeer, S., Rajamani, S.K.: Decomposing refinement proofs using assume-guarantee reasoning. In: Sentovich, E. (ed.) International Conference on Computer Aided Design (ICCAD), pp. 245–252. IEEE, Piscataway (2000)
52. Henzinger, T.A., Qadeer, S., Rajamani, S.K., Tasiran, S.: An assume-guarantee rule for checking simulation. In: Gopalakrishnan, G., Windley, P.J. (eds.) Formal Methods in Computer Aided Design (FMCAD). LNCS, vol. 1522, pp. 421–432. Springer, Heidelberg (1998)
53. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
54. Howar, F., Giannakopoulou, D., Rakamaric, Z.: Hybrid learning: interface generation through static, dynamic, and symbolic analysis. In: Intl. Symp. on Software Testing and Analysis (ISSTA), pp. 268–279. ACM, New York (2013)
55. Inverardi, P., Wolf, A.L., Yankelevich, D.: Static checking of system behaviors using derived component assumptions. Trans. Softw. Eng. Methodol. **9**(3), 239–272 (2000)
56. Ip, C.N., Dill, D.L.: Better verification through symmetry. In: Agnew, D., Claesen, L.J.M., Camposano, R. (eds.) CHDL. IFIP Transactions, vol. A-32, pp. 97–111. North-Holland, Amsterdam (1993)
57. Jones, C.B.: Tentative steps toward a development method for interfering programs. Trans. Program. Lang. Syst. **5**(4), 596–619 (1983)
58. Komuravelli, A., Pasareanu, C.S., Clarke, E.M.: Learning probabilistic systems from tree samples. In: Symp. on Logic in Computer Science, vol. LICS, pp. 441–450. IEEE, Piscataway (2012)
59. Kurshan, R.: Reducibility in analysis of coordination. In: Varaiya, P., Kurzhanski, A. (eds.) Discrete Event Systems: Models and Applications. Lecture Notes in Control and Information Sciences, vol. 103, pp. 19–39. Springer, Heidelberg (1988)
60. Lamport, L.: Proving the correctness of multiprocess programs. Trans. Softw. Eng. **3**(2), 125–143 (1977)

61. Lamport, L.: Composition: a way to make proofs harder. In: de Roever, W.P., Langmaack, H., Pnueli, A. (eds.) Compositionality: The Significant Difference (COMPOS '97). LNCS, vol. 1536, pp. 402–423. Springer, Heidelberg (1998)
62. Larsen, K., Xinxin, L.: Equation solving using modal transition systems. In: Symp. on Logic in Computer Science, vol. LICS, pp. 108–117. IEEE, Piscataway (1990)
63. Lin, S.W., André, É., Liu, Y., Sun, J., Dong, J.S.: Learning assumptions for compositional verification of timed systems. Trans. Softw. Eng. **40**(2), 137–153 (2014)
64. Magee, J., Kramer, J.: Concurrency: State Models & Java Programs. Wiley, New York (1999)
65. Maier, P.: Compositional circular assume-guarantee rules cannot be sound and complete. In: Gordon, A.D. (ed.) Intl. Conf. on Foundations of Software Science and Computational Structures (FoSSaCS). LNCS, vol. 2620, pp. 343–357. Springer, Heidelberg (2003)
66. McMillan, K.L.: Verification of an implementation of Tomasulo's algorithm by compositional model checking. In: Hu, A.J., Vardi, M.Y. (eds.) Intl. Conf. on Computer-Aided Verification (CAV), vol. 1427, pp. 110–121. Springer, Heidelberg (1998)
67. McMillan, K.L.: Circular compositional reasoning about liveness. In: Pierre, L., Kropf, T. (eds.) Correct Hardware Design and Verification Methods (CHARME). LNCS, vol. 1703, pp. 342–345. Springer, Heidelberg (1999)
68. Merlin, P., Bochmann, G.V.: On the construction of submodule specifications and communication protocols. Trans. Program. Lang. Syst. **5**(1), 1–25 (1983)
69. Misra, J., Chandy, K.M.: Proofs of networks of processes. Trans. Softw. Eng. **7**(4), 417–426 (1981)
70. Moon, I.H., Kukula, J.H., Shiple, T.R., Somenzi, F.: Least fixpoint approximations for reachability analysis. In: White and Sentovich [86], pp. 41–44
71. Nam, W., Madhusudan, P., Alur, R.: Automatic symbolic compositional verification by learning assumptions. Form. Methods Syst. Des. **32**(3), 207–234 (2008)
72. Namjoshi, K.S.: Symmetry and completeness in the analysis of parameterized systems. In: Cook, B., Podelski, A. (eds.) Intl. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI). LNCS, vol. 4349, pp. 299–313 (2007)
73. Namjoshi, K.S., Trefler, R.J.: On the completeness of compositional reasoning methods. Trans. Comput. Log. **11**(3), 16:1–16:22 (2010)
74. Namjoshi, K.S., Trefler, R.J.: Local symmetry and compositional verification. In: Kuncak, V., Rybalchenko, A. (eds.) Intl. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI). LNCS, vol. 7148, pp. 348–362 (2012)
75. Namjoshi, K.S., Trefler, R.J.: Uncovering symmetries in irregular process networks. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) Intl. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI). LNCS, vol. 7737, pp. 496–514 (2013)
76. Owicki, S.S., Gries, D.: Verifying properties of parallel programs: an axiomatic approach. Commun. ACM **19**(5), 279–285 (1976)
77. Papadimitriou, C.H.: Computational Complexity. Addison-Wesley, Reading (1994)
78. Pasareanu, C.S., Giannakopoulou, D., Bobaru, M.G., Cobleigh, J.M., Barringer, H.: Learning to divide and conquer: applying the L* algorithm to automate assume-guarantee reasoning. Form. Methods Syst. Des. **32**(3), 175–205 (2008)
79. Piterman, N., Pnueli, A.: Temporal logic and fair discrete systems. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
80. Pnueli, A.: In transition from global to modular temporal reasoning about programs. In: Apt, K. (ed.) Logics and Models of Concurrent Systems, pp. 123–144. Springer, Heidelberg (1985)
81. Pnueli, A., Ruah, S., Zuck, L.: Automatic deductive verification with invisible invariants. In: Margaria, T., Yi, W. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 2031, pp. 82–97. Springer, Heidelberg (2001)
82. Rivest, R.L., Schapire, R.E.: Inference of finite automata using homing sequences. Inf. Comput. **103**(2), 299–347 (1993)

83. de Roever, W.P., de Boer, F.S., Hannemann, U., Hooman, J., Lakhnech, Y., Poel, M., Zwiers, J.: Concurrency Verification: Introduction to Compositional and Noncompositional Methods. Cambridge Tracts in Theoretical Computer Science, vol. 54. Cambridge University Press, Cambridge (2001)
84. Scott, D., Strachey, C.: Toward a mathematical semantics for computer languages. Tech. Rep. PRG-6, Oxford Programming Research Group (1971)
85. Singh, R., Giannakopoulou, D., Pasareanu, C.S.: Learning component interfaces with may and must abstractions. In: Touili, T., Cook, B., Jackson, P. (eds.) Intl. Conf. on Computer-Aided Verification (CAV), vol. 6174, pp. 527–542. Springer, Heidelberg (2010)
86. White, J.K., Sentovich, E. (eds.): Proceedings of the 1999 IEEE/ACM International Conference on Computer-Aided Design, 1999, San Jose, California, USA, November 7–11, 1999. IEEE, Piscataway (1999)
87. Xiao, H., Sun, J., Liu, Y., Lin, S.W., Sun, C.: Tzuyu: learning stateful typestates. In: Denney, E., Bultan, T., Zeller, A. (eds.) Intl. Conf. on Automated Software Engineering (ASE), pp. 432–442. IEEE, Piscataway (2013)

Chapter 13
Abstraction and Abstraction Refinement

Dennis Dams and Orna Grumberg

Abstract Abstraction, in the context of model checking, is aimed at reducing the state space of the system by omitting details that are irrelevant to the property being verified. Many successful approaches to the "state explosion problem," some of them described in other chapters, can be seen as abstractions. In this chapter, several notions of abstraction are considered in a uniform setting. Different such notions lead to a variety of preservation results that establish which kind of temporal properties may be verified via an abstracted system. We first define the needed background on simulation and bisimulation relations and their logic preservation. We then present the abstraction that is currently most widely used in practice: existential abstraction, which preserves universal fragments of branching-time logics. We give examples of such abstractions: localization reduction for hardware and predicate abstraction for software. We then proceed to stronger abstractions which preserve full branching-time logics. We introduce Kripke Modal Transition Systems and modal simulation, and show logic preservation. We close the chapter with a review of the presented results in the light of the notion of completeness.

13.1 Introduction

On Wikipedia [101], *abstraction* is defined as follows:

> Abstraction is a process by which higher concepts are derived from the usage and classification of literal ("real" or "concrete") concepts, first principles and/or other abstractions. An "abstraction" (noun) is a concept that acts as super-categorical noun for all subordinate concepts, and connects any related concepts as a group, field, or category.
> Abstractions may be formed by reducing the information content of a concept or an observable phenomenon, typically to retain only information which is relevant for a particular purpose. For example, abstracting a leather soccer ball to the more general idea of a ball retains only the information on general ball attributes and behavior, eliminating the characteristics of that particular ball.

D. Dams
Embedded Systems Innovation by TNO, Eindhoven, The Netherlands

O. Grumberg (✉)
Technion, Israel Institute of Technology, Technion City, Haifa, Israel
e-mail: orna@cs.technion.ac.il

E.M. Clarke et al. (eds.), *Handbook of Model Checking*,
DOI 10.1007/978-3-319-10575-8_13

In the first paragraph, apart from the technical terms which are not relevant to this chapter, it is clarified that the word *abstraction* is used to denote both the process of abstracting as well as the possible result of that process. In this chapter, we also use the term *abstract object* to denote the latter meaning.

According to the second paragraph, the process of abstraction starts from a "concept or an observable phenomenon," which we will call a *concrete object*. The process yields an object in which only certain information is retained; the other information that is present in the concrete object is sometimes said to have been *abstracted away*. For example, the integer number -3 can be abstracted to its sign, which can, for example, be denoted $-$, or **neg**, or $\{\ldots, -4, -3, -2, -1\}$. Its absolute value is thus abstracted away; its sign is the resulting abstract object. Sometimes, concrete and abstract objects look the same, in the sense that from their representation alone one cannot tell whether an object is concrete or abstract. As an example, consider rectangles in a graphical drawing application that have a fill pattern. Suppose that these rectangles are abstracted by removing their fill pattern, and that the resulting rectangles are shown with a white interior. When seeing such a white rectangle, one cannot tell whether it is a concrete rectangle whose fill pattern is "uniform white," or an abstract rectangle. To avoid such confusion, when representations of concrete and abstract objects look the same, we will nevertheless consider the domains of concrete and of abstract objects to be of different types.

In this chapter, we consider the abstraction of Kripke structures, focusing on finding suitable concepts to use as abstractions, and also on the process of abstraction. We reduce the information content of Kripke structures with the particular purpose of making it easier to model check temporal logic properties on them. In the statement of a model-checking problem, the Kripke structure is usually not given directly, but, for example, as a software system represented in some programming language; the intended Kripke structure is then obtained as a particular semantic interpretation of that system description. Constructing and representing that Kripke structure in a way that allows efficient checking of temporal logic properties forms one of the main challenges of the field of model checking. It is often impossible to store a "naive" representation of such a Kripke structure (e.g., by explicitly listing the states and transitions) on a computer, because of its size. This is called the *state explosion problem*. Solutions to this problem based on symbolic representations of the Kripke structure are presented elsewhere in this handbook [10, 21] (Chaps. 8 and 10). In this chapter we take a different approach, that of abstraction.

Abstraction tackles this challenge based on the assumption that a reduction of the information content results in a reduction of the size of the representation of a Kripke structure. Given a temporal property (or a set of them) to be checked, information that is not relevant to the valuation of any of those properties can be omitted from the Kripke structure. The resulting abstraction need not be a Kripke structure itself, but it should come with a notion of evaluating a temporal property on it. This chapter presents several possible choices for such abstractions, explaining for each of them how they relate to the concrete Kripke structures that they abstract, and how to evaluate temporal properties on them.

Besides the issue of what kind of object the abstraction of a Kripke structure should be, there is the question of how to *construct* that object. Given a notion of

abstraction, e.g., defined via a function α mapping a Kripke structure to its abstraction, in practice, the latter usually cannot be constructed by first constructing the concrete Kripke structure from the system description and then applying α to it. This would defy the purpose of avoiding the creation of the (potentially too big or even infinite) concrete Kripke structure. Instead, abstractions are built by applying "non-standard" semantic interpretations to system descriptions. The CEGAR approach (CounterExample-Guided Abstraction Refinement) discussed later in this chapter, and elsewhere in this Handbook [65] (Chap. 15), can be seen as a collection of algorithms for this kind of construction. The topic of constructing abstractions is also one of the focuses of the theory of *Abstract Interpretation* [8, 37–40, 76], which is not treated in this chapter.

13.2 Preliminaries

13.2.1 Abstraction Frameworks

To formalize some of the terms introduced above, we use the notion of an *abstraction framework*.

Definition 1 An *abstraction framework* is a tuple $(C, L, [\![\cdot]\!], A, \rho, [\![\cdot]\!]^\alpha)$, where the components are as follows. The set C contains the *concrete objects*: the things whose properties we are interested in, e.g., Kripke structures. Properties are expressed as formulas in a *logic* L and the interpretation of $\varphi \in L$ w.r.t. concrete objects is denoted $[\![\varphi]\!]$, which is the set of concrete objects for which φ is true. A is the set of *abstract objects* (or *abstractions*), which reduce the information content of concrete objects, e.g. in order to render them amenable to automated techniques for property checking. The *abstraction relation* $\rho \subseteq C \times A$ specifies how each concrete object can be abstracted. That is, for a given $c \in C$, all those $a \in A$ such that $\rho(c, a)$ are abstractions of c. The interpretation $[\![\varphi]\!]^\alpha$ is the set of abstract objects for which φ holds.[1]

It is helpful to keep in mind the view that any abstract object a represents a *set of* concrete objects, namely those for which a is an abstraction: $\{c \in C \mid \rho(c, a)\}$. This set is called the *concretization* of a, denoted $\gamma(a)$.

The truth values *true* and *false* are denoted tt and ff respectively. For a set S of objects (e.g., $S = [\![\varphi]\!]$ or $S = [\![\varphi]\!]^\alpha$), we often treat S as a predicate symbol that denotes the predicate whose characteristic set is S. Thus, for a single (concrete or abstract) object x we then say that $S(x)$ holds (writing $S(x) = \text{tt}$ or even just $S(x)$) to mean that $x \in S$. Furthermore, in the notation $[\![\varphi]\!]^\alpha$ we drop the superscript α

[1] One could choose different logics on the concrete and abstract sides, but this would unnecessarily complicate the discussion here.

when the type of the elements is clear. For example, if a is an abstract object, we write $[\![\varphi]\!](a)$ instead of $[\![\varphi]\!]^\alpha(a)$.

A principal requirement for any abstraction framework is that it is *sound*.

Definition 2 An abstraction framework $(C, L, [\![\cdot]\!], A, \rho, [\![\cdot]\!]^\alpha)$ is *sound* if whenever $\rho(c, a)$ and $[\![\varphi]\!]^\alpha(a) = \mathrm{tt}$, then $[\![\varphi]\!](c) = \mathrm{tt}$.

Soundness ensures that we can establish properties of a concrete object by inspecting an abstraction of it. Soundness as stated here is also called *soundness for true*, since the premise requires $[\![\varphi]\!]^\alpha(a)$ to be true. A stronger soundness requirement, considered later on in this chapter, extends to "soundness for both true and false."

13.2.2 Kripke Structures

To represent the semantics of programs, we use Kripke structures as defined elsewhere in this Handbook [88] (Chap. 3), albeit with slightly different notational conventions. Kripke structures will play the role of concrete objects (the set C from above) throughout this chapter.

Definition 3 A *Kripke structure* $M = (AP, S, I, R, [\![\cdot]\!])$ consists of:

- a set AP of *propositional symbols* (or *atomic propositions*),
- a set S of *states*,
- a set $I \subseteq S$ of *initial states*,
- a *transition relation* $R \subseteq S \times S$, and
- a *propositional interpretation* $[\![\cdot]\!] : AP \to S \to \{\mathrm{ff}, \mathrm{tt}\}$.

$[\![\cdot]\!]$ turns the symbols in AP into predicates over states in S. So, $[\![p]\!](s) = \mathrm{tt}$ (ff) says that p is true (false) in state s. When $[\![\cdot]\!]$ is clear from the context, we sometimes write $p(s)$ instead of $[\![p]\!](s)$.

An alternative definition [31] includes a *labeling function* $L : S \to 2^{AP}$ as part of the Kripke structure, instead of $[\![\cdot]\!]$. In that case, for each state $s \in S$, $L(s)$ is the set of all atomic propositions which are true in s; all propositions in $AP \setminus L(s)$ are then false in s.

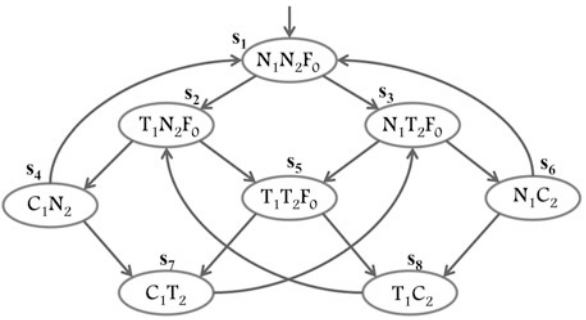
We fix a set AP of propositional symbols throughout this chapter, with typical elements p, q. A *literal* is a proposition from AP or its negation (denoted $\neg p$, for $p \in AP$).

Example 1 Consider a simple program which implements a mutual exclusion algorithm for two processes, P_1 and P_2, running in parallel. The processes share a Boolean variable *Flag*, indicating whether getting into the critical section is permitted. Each process has a local variable v_i (i being 1 or 2) ranging over

Fig. 1 Process P_i

$$
\begin{array}{lcl}
v_i = \mathit{Neutral} & \rightarrow & v_i := \mathit{Trying} \\
v_i = \mathit{Trying} \ \wedge \ \mathit{Flag} = \mathrm{tt} & \rightarrow & v_i := \mathit{Critical}; \ \ \mathit{Flag} := \mathrm{ff} \\
v_i = \mathit{Critical} & \rightarrow & v_i := \mathit{Neutral}; \ \ \mathit{Flag} := \mathrm{tt}
\end{array}
$$

Fig. 2 Kripke structure M for the mutual exclusion program



{*Neutral*, *Trying*, *Critical*}, indicating whether process P_i is outside (neutral state), trying to get into, or inside its critical section. The program is depicted in Fig. 1. It is a "do forever" loop with three conditional choices, where conditions appear to the left of the arrows and pieces of code to be executed appear on the right.

The Kripke structure M representing the global behavior of the program is given in Fig. 2. Its set of states is the set of all combinations of valuations of v_1, v_2, and *Flag*. That is, $S = \{\mathit{Neutral}, \mathit{Trying}, \mathit{Critical}\} \times \{\mathit{Neutral}, \mathit{Trying}, \mathit{Critical}\} \times \{\mathrm{tt}, \mathrm{ff}\}$. For simplicity, only states which are reachable from the initial state are shown in the figure. Its initial state, indicated by an incoming arrow, is $s_1 = (\mathit{Neutral}, \mathit{Neutral}, \mathrm{tt})$. The transitions are shown by arrows between states. Note that the transitions assume a semantics in which each of the program's choices is executed in an atomic manner. The set of atomic propositions is $AP = \{N_1, T_1, C_1, N_2, T_2, C_2, F_0\}$. The atomic propositions are interpreted as follows. For every $s \in S$,

$$
\begin{aligned}
[\![N_i]\!](s) \quad &\text{iff} \quad s(v_i) = \text{Neutral}, \\
[\![T_i]\!](s) \quad &\text{iff} \quad s(v_i) = \text{Trying}, \\
[\![C_i]\!](s) \quad &\text{iff} \quad s(v_i) = \text{Critical}, \\
[\![F_0]\!](s) \quad &\text{iff} \quad s(\mathit{Flag}) = \mathrm{tt}.
\end{aligned}
$$

In Fig. 2, each state s is labeled with the set of atomic propositions from AP that are true in s. Atomic propositions from AP which do not label s are false in s.

The CTL formula[2] $\forall \mathsf{G} \neg (C_1 \wedge C_2)$ specifies the property of mutual exclusion: It says that in none of the reachable states are both processes in their critical section. It is easy to see that this property holds in M. The formula $\forall \mathsf{G}(T_i \rightarrow \forall \mathsf{F} C_i)$ specifies non-starvation: If a process is trying to get into its critical section, it eventually will get in. This property does not hold. For example, process 1 "starves" along the path $s_1, (s_2, s_5, s_8)^\omega$. □

[2] See elsewhere in this Handbook [84] (Chap. 2) and also below.

It is sometimes useful to restrict the discussion to *finite state* Kripke structures, where S is finite, or to structures in which R is *total*, that is, for every $s \in S$ there is $s' \in S$ such that $(s, s') \in R$.

A *path* in M from a state s is a non-empty sequence of states, $\pi = s_0, s_1, \ldots$, such that $s = s_0$ and for every $i \geq 0$, if s_i is not the last state in π then $(s_i, s_{i+1}) \in R$. When s_i is the last element of π, we define $length(\pi) = i$. If π is infinite, then $length(\pi) = \infty$ where ∞ is greater than any natural number i. We write $\pi[i]$ for s_i. A path π is *maximal* if it is infinite or if its last state has no successor in R.

It is straightforward to represent a *fairness-free* Fair Discrete System (FDS, defined elsewhere in this Handbook [84] (Chap. 2)) $\mathscr{D} = \langle \mathscr{V}, \Theta, \rho \rangle$ by a Kripke structure $(AP, S, I, R, [\![\cdot]\!])$. Let $\mathscr{V} = \{v_1, \ldots, v_n\}$ where each variable v_i is defined over domain D_i. Then for the set S of states of the Kripke structure, take $D_1 \times \cdots \times D_n$, the set of all possible valuations for $\mathscr{V}$. For R, take $\{(s, s') \mid (s, s') \models \rho\}$; and let $I = \{s \mid s \models \Theta\}$. Further, the atomic propositions in AP are the state formulas associated with the FDS, with $[\![\cdot]\!]$ being the usual interpretation (see Chap. 2 for details).

The general results presented in this chapter do not depend on the internal structure of individual states as long as a valuation of the atomic propositions over states is given. In particular, the assumption that states are valuations of variables, as is the case in Fair Discrete Systems, is not needed. Another reason to use Kripke structures instead of FDSs is that the fairness requirements of an FDS are relative to *paths*, and they are used to restrict the valuation of temporal formulas to those paths that are fair. This makes sense for temporal logics that refer to paths explicitly via (universal and existential) *path quantifiers*, such as CTL* and its sublogics. In this chapter, we use the μ-calculus, introduced briefly below and presented in more detail elsewhere in this Handbook [14] (Chap. 26), which does not explicitly refer to paths. As a result, the valuation of μ-calculus formulas is not naturally defined over objects that come with a notion of fair paths.[3]

13.2.3 The μ-Calculus

To express properties of Kripke structures, in this chapter we use the μ-calculus, $\mathscr{L}_\mu$ [67]. We refer the reader to Chap. 26 for a detailed account of the μ-calculus. Here, we give a somewhat simplified definition. Note that our Kripke structures do not have actions labeling the transitions; therefore, we can omit the action symbols from the μ-calculus modal operators, writing them as $\Box$ ("for all next states") and $\Diamond$ ("there exists a next state").

Definition 4 The syntax of the μ-calculus, $\mathscr{L}_\mu$, is defined relative to the given set AP of propositional symbols. A formula is built up from literals (propositional

[3] Path fairness can be captured in the μ-calculus by encoding it as part of the formula.

symbols and their negations), the logical connectives $\vee$ and $\wedge$, the modal operators $\Diamond$ and $\Box$, and the fixpoint operators μ and ν, as defined in Chap. 26. The fixpoint operators depend on variables (typically X, Y).

Note that negation is only allowed on propositions. Thus the formulas of $\mathscr{L}_\mu$ as defined above are in negation normal form.[4] In examples, we will use general negation. As the term "normal form" already suggests, a formula using general negation can be brought into negation normal form. This is done by "pushing the negations inward," while exchanging operators with their dual ones. For instance, $\wedge$ is exchanged with $\vee$; $\Box$ with $\Diamond$; μ with ν; F with G; $\forall$ with $\exists$; and conversely. Also, we use $\rightarrow$ for logical implication, defined in the usual way ($\varphi_1 \rightarrow \varphi_2$ is defined as $\neg\varphi_1 \vee \varphi_2$). We also use the syntax of CTL* (see Chap. 2 [84]) in many places throughout this chapter. Whenever a formula is written in CTL* syntax, it is assumed to denote an equivalent formula in the μ-calculus. The latter always exists, as explained in Chap. 26 [14], specifically in Sect. 26.4.2.

In this chapter we present various abstraction frameworks; in each of these, the logic (the component L in Definition 1 from Sect. 13.2.1) is $\mathscr{L}_\mu$ or a fragment of it.

Definition 5 The *universal fragments* of the logics $\mathscr{L}_\mu$ and CTL/CTL* are defined as follows:

- Let φ be a $\mathscr{L}_\mu$ formula in negation normal form. Then, φ is a *universal* $\mathscr{L}_\mu$ formula if it does not contain the modal operator $\Diamond$ (in other words, if φ contains any modal operators, these must all be $\Box$). $\Box\mathscr{L}_\mu$ is the set of all universal $\mathscr{L}_\mu$ formulas.
- Let φ be a CTL* formula in negation normal form. Then φ is a *universal* CTL* formula if it does not contain the existential path quantifier $\exists$. ACTL* is the set of all universal CTL* formulas. The universal fragment of CTL, denoted ACTL, is defined similarly.

Note that the universal fragments include both liveness operators (μ, F) and safety operators (ν, G).

The following definition specifies how to evaluate $\varphi \in \mathscr{L}_\mu$ over a Kripke structure $M = (AP, S, I, R, [\![\cdot]\!])$, i.e., to determine whether M satisfies the property expressed by φ. In this case it is said that "M is a model of φ." This is done by extending the function $[\![\cdot]\!]$ which specifies how to evaluate propositions from AP over individual states of M, as follows. First, it is extended to all formulas of $\mathscr{L}_\mu$, so that $[\![\varphi]\!]$ denotes the set of all states of M for which φ holds. Since the property expressed by φ now possibly depends not only on s, but also on other states in M that are reachable from s, the definition of $[\![\varphi]\!]$ depends on the transition relation R of M. Second, it is extended to evaluate a formula over a Kripke structure as a

[4] Another term used is *positive normal form*.

whole, i.e., $[\![\varphi]\!](M)$ is defined. As expected, this is where the initial states of M play a role.

Definition 6 Let $M = (AP, S, I, R, [\![\cdot]\!])$ be a Kripke structure. The definition of $[\![\cdot]\!] : AP \to S \to \{\text{tt}, \text{ff}\}$ is extended to $[\![\cdot]\!] : \mathscr{L}_\mu \to S \to \{\text{tt}, \text{ff}\}$, as follows. Here, ψ, ψ_1, and ψ_2 are formulas from $\mathscr{L}_\mu$.

$$
\begin{array}{lll}
[\![\neg\psi]\!](s) = \text{tt} & \text{iff} & [\![\psi]\!](s) = \text{ff} \\
[\![\psi_1 \wedge \psi_2]\!](s) = \text{tt} & \text{iff} & [\![\psi_1]\!](s) = \text{tt} \text{ and } [\![\psi_2]\!](s) = \text{tt} \\
[\![\psi_1 \vee \psi_2]\!](s) = \text{tt} & \text{iff} & [\![\psi_1]\!](s) = \text{tt} \text{ or } [\![\psi_2]\!](s) = \text{tt} \\
[\![\Box\psi]\!](s) = \text{tt} & \text{iff} & \forall t \in S : R(s,t) \text{ implies } [\![\psi]\!](t) = \text{tt} \\
[\![\Diamond\psi]\!](s) = \text{tt} & \text{iff} & \exists t \in S : R(s,t) \text{ and } [\![\psi]\!](t) = \text{tt} \\
[\![\mu X.\psi(X)]\!](s) = \text{tt} & \text{iff} & \forall S' \subseteq S : [\![\psi(S')]\!] = S' \text{ implies } s \in S' \\
[\![\nu X.\psi(X)]\!](s) = \text{tt} & \text{iff} & \exists S' \subseteq S : [\![\psi(S')]\!] = S' \text{ and } s \in S'
\end{array}
$$

A note about the notation in the last two cases, which are somewhat informal: For a variable X (bound by ν or μ), the notation $\psi(X)$ is used for a subformula in which X occurs free. In the context of this notation, $[\![\psi(S')]\!]$ (where S' is a set of states) denotes the evaluation of $\psi(X)$ in which X is evaluated to S', i.e., in which $[\![X]\!] = S'$. This can be further formalized by carrying the evaluation of free variables around as an extra argument in the definition, as done elsewhere in this Handbook [14] (Chap. 26). Here, it would unnecessarily complicate the notation.

For $\varphi \in \mathscr{L}_\mu$, $[\![\varphi]\!](M)$ is defined to be true if and only if $[\![\varphi]\!](s)$ is true for all $s \in I$ (i.e., for all initial states of M).

13.2.3.1 Some Notes on the Difference Between $\mathscr{L}_\mu$ and CTL/CTL*

The $\mathscr{L}_\mu$ modalities $\Diamond$ and $\Box$ in the current chapter are quite different from what those symbols represent elsewhere in this Handbook [84] (Chap. 2). There they are (derived) operators that express the notions "eventually" and "globally," respectively; those notions are denoted by the operators F and G in the current chapter. The $\mathscr{L}_\mu$ modalities $\Diamond$ and $\Box$ may be viewed as branching-time generalizations of the "next" operator $\bigcirc$ from Chap. 2, where a computation state always has a single, unique successor state, to the case where a state may have zero, one, or multiple successors. The LTL (Linear Temporal Logic) formula $\bigcirc p$ means "in the next state (on the current computation) p holds." The $\mathscr{L}_\mu$ formula $\Diamond p$ means "there exists a successor state (in the Kripke structure) in which p holds," and $\Box p$ means "in all successor states, p holds."

Thus, the $\mathscr{L}_\mu$ modalities $\Diamond$ and $\Box$ carry the existential and universal character that is expressed in CTL and CTL* by the quantifiers $\exists$ and $\forall$. An important difference is that the latter quantify over the computations (infinite paths of a Fair Discrete Transition System) that originate at the current state, whereas $\Diamond$ and $\Box$ only talk about the successor states (if any) of the current state in a Kripke structure. In order to express quantification over whole computations in $\mathscr{L}_\mu$, the "one-step-deep" $\Diamond$ and $\Box$ are combined with iteration as expressed by μ (bounded iteration, for eventuality,

e.g., "p will hold within a finite number of steps") and ν (unbounded iteration, for invariance, e.g., "p always holds, or, in other words, it does not eventually happen that p does not hold"). For example, consider the CTL formula $\exists \mathsf{G} p$, expressing that from the current state, there exists a computation on which always (globally) p holds. An equivalent formula in $\mathscr{L}_\mu$ is $\nu X.p \wedge \Diamond X$. The $\Diamond$ modality expresses existential quantification over next states, but not beyond them; in order to reach the effect of existentially quantifying over computations, it is iterated by adding a fixpoint operator around it. Since the number of iterations is infinite (p must continue to hold, no matter the number of steps taken), the greatest fixpoint operator ν is used.

When restricting logics to universal or existential fragments, some care has to be taken. The point is that a formula in ACTL*, the universal fragment of CTL*, may contain a $\Diamond$ when translated into $\mathscr{L}_\mu$, and therefore it will not be universal in $\mathscr{L}_\mu$. The reason is that the eventuality F of (A)CTL* has "an existential flavor" to it, when interpreted on finite paths. This is again best illustrated by an example. The ACTL* formula $\forall \mathsf{F} p$ says that along all computations that originate from the current state, p must eventually hold. When we break this up into iterated requirements about successor states, it becomes something like: "either [p holds in the current state], or [there must exist at least one successor, and furthermore in all successor states, the requirement holds recursively]; and we do not allow infinite postponement of p." More formally, we have the following equivalence:

$$\forall \mathsf{F} p \equiv \big(p \vee (\exists \mathsf{X} \mathit{true} \wedge \forall \mathsf{X} \forall \mathsf{F} p)\big)$$

In $\mathscr{L}_\mu$, the formula becomes:

$$\mu X.p \vee (\Diamond \mathit{true} \wedge \Box X)$$

The part "there must exist at least one successor" (which shows up in the formulas as $\exists \mathsf{X} \mathit{true}$ and $\Diamond \mathit{true}$ resp.) is needed because otherwise the formula might be fulfilled vacuously when the current state does not satisfy p and has no successors; this is because any formula of the form $\Box\varphi$ will be true in any state that has no successors. In the ACTL* formula $\forall \mathsf{F} p$, this requirement is captured implicitly by the fact that such a formula must be interpreted over a *computation*, in which there is always a (unique) successor.

If the transition relation R of the Kripke structure is *total*, i.e., if for every $s \in S$ there is $s' \in S$ such that $(s, s') \in R$, then $\exists \mathsf{X} \mathit{true}$ holds in every state, and therefore in that case we have:

$$\forall \mathsf{F} p \equiv (p \vee \forall \mathsf{X} \forall \mathsf{F} p) \equiv \mu X.p \vee \Box X$$

Thus, $\forall \mathsf{F}$ and also $\forall \mathsf{X}$ and $\forall \mathsf{U}$ have a universal character on models with total transition relations. However, on models with non-total transition relations they have a combination of existential and universal characteristics.

13.3 Simulation and Bisimulation Relations

In this section we define two of the most commonly used relations over Kripke structures. We also present the logic preservation they induce. *Bisimulation* is an *equivalence relation* which roughly says that two Kripke structures have the same behaviors, even though their sets of states and transitions may differ. *Simulation* is a *preorder* in which the larger structure may have more behaviors, but possibly fewer states and transitions. Bisimulation guarantees the preservation of full branching-time logics, while simulation guarantees the preservation of only the universal fragment of such logics.

The two relations can be exploited in model checking as follows: Instead of checking the concrete (full) model of a system we will check a related, reduced model with fewer states and transitions. Thus, applying model checking to it is easier. By the preservation theorems presented below we will be able to conclude that if the reduced structure is bisimilar to the concrete one than every property true (false) in the former is also true (false) in the latter. On the other hand, if the reduced structure simulates the concrete one then only truth of *universal* properties can be concluded.

We notice that, in the context of abstraction, bisimulation is less desirable: Insisting on preservation of both truth and falsity of properties implies that less information can be abstracted away, and thus less reduction can be achieved.

13.3.1 Simulation Relation

We start by defining the simulation preorder over Kripke structures [81]. Simulation between two models is checked state-wise: One state is smaller than another by the simulation relation if they agree on their common atomic propositions, and if for every successor of the smaller state there is a corresponding successor of the greater one. Formally, we have the following.

Definition 7 Let $M_1 = (AP_1, S_1, I_1, R_1, [\![\cdot]\!]_1)$ and $M_2 = (AP_2, S_2, I_2, R_2, [\![\cdot]\!]_2)$ be Kripke structures, such that $AP_2 \subseteq AP_1$. A relation $H \subseteq S_1 \times S_2$ is a *simulation relation [81] from* M_1 *to* M_2 if for every $s_1 \in S_1$ and $s_2 \in S_2$ such that $H(s_1, s_2)$, both of the following conditions hold:

- For all $p \in AP_2$, $[\![p]\!]_1(s_1)$ iff $[\![p]\!]_2(s_2)$ and
- $\forall\, t_1\ [[\ R_1(s_1, t_1)] \Rightarrow \exists\, t_2\ [[\ R_2(s_2, t_2) \wedge H(t_1, t_2)\ ]]]$

We write $s_1 \leq s_2$ for $H(s_1, s_2)$. We say that M_1 *is simulated by* M_2 (or M_2 *simulates* M_1) (denoted $M_1 \leq M_2$) if there exists a simulation relation H from M_1 to M_2 such that

$$\forall\, s_1 \in I_1 \left[\exists\, s_2 \in I_2 \left[H(s_1, s_2)\right]\right].$$

Example 2 Consider the Kripke structure $M_1 = (AP_1, S_1, I_1, R_1, [\![\cdot]\!]_1)$ in Fig. 3, where $AP_1 = \{C_1, C_2, F_0\}$, $S_1 = \{a_1, a_2, a_3\}$, and $I_1 = \{a_1\}$. The transition relation

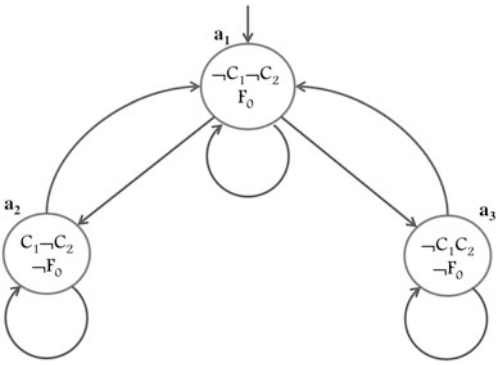


Fig. 3 Abstract Kripke structure M_1 for the mutual exclusion program

and the state labeling are shown in the figure, where state labeling includes negation of atomic propositions explicitly.

Also consider the Kripke structure M of Fig. 2. In order to show that $M \leq M_1$ we first notice that $AP_1 \subseteq AP$. A simulation relation $H \subseteq S \times S_1$ from M to M_1 can be defined as follows:

$$H = \{(s_1, a_1), (s_2, a_1), (s_3, a_1), (s_4, a_2), (s_5, a_1), (s_6, a_3), (s_7, a_2), (s_8, a_3)\}.$$

It is easy to check that corresponding states agree on the common atomic propositions, and that for every successor of a state in M there is a corresponding successor of the simulating state in M_1.

For example, $(s_2, a_1) \in H$ and for the successors s_4, s_5 of s_2 it holds that $(s_4, a_2) \in H$ and $(s_5, a_1) \in H$, where a_2, a_1 are successors of a_1. □

The simulation relation induces a relationship between paths in M_1 and M_2, as described in the following lemma.

Lemma 1 *Let H be a simulation relation from M_1 to M_2 and assume that $s_1 \in S_1$, $s_2 \in S_2$, and $H(s_1, s_2)$. Then, for every maximal path π_1 from s_1 there is a maximal path π_2 from s_2 such that $length(\pi_1) \leq length(\pi_2)$. Further, for every $i \leq length(\pi_1)$, $H(\pi_1[i], \pi_2[i])$.*

The following theorem states the preservation of universal formulas in the μ-calculus and in CTL* and CTL.

Theorem 1 *Let $M_1 \leq M_2$, $s_1 \leq s_2$, and let φ be a formula with atomic propositions in AP_2. Then, the following holds*:

1. [40, 76] *Assume that φ is in $\Box\mathscr{L}_\mu$. Then $[\![\varphi]\!]_2(s_2)$ implies $[\![\varphi]\!]_1(s_1)$ and $[\![\varphi]\!]_2(M_2)$ implies $[\![\varphi]\!]_1(M_1)$.*
2. [51] *Assume that R_1 is total and φ is in ACTL*. Then $[\![\varphi]\!]_2(s_2)$ implies $[\![\varphi]\!]_1(s_1)$ and $[\![\varphi]\!]_2(M_2)$ implies $[\![\varphi]\!]_1(M_1)$.*

Note that the result holds for ACTL and LTL formulas as well. Also, as mentioned above, the universal fragments preserved by the simulation relation include both safety and liveness properties.

Consider Example 2 again. Consider also the ACTL property $\varphi = \forall \mathsf{G}(F_0 \leftrightarrow (\neg C_1 \wedge \neg C_2))$. Since $M_1 \models \varphi$, by Theorem 1 we conclude that $M \models \varphi$ as well.

The requirement in Item 2 that R_1 must be total is necessary for preservation of ACTL*. This is because if R_1 is not total then a path of M_1 may correspond to just a prefix of a path in M_2. Assume $M_2 \models \forall \mathsf{F} p$. Then, M_1 may fail to satisfy $\forall \mathsf{F} p$ due to a "short" path that does not reach a state satisfying p. The discussion above demonstrates itself also in the equivalences discussed in Sect. 13.2.3.1.

13.3.2 Bisimulation Relation

We next define the bisimulation equivalence [83] over Kripke structures. Bisimulation between two models is also checked state-wise: One state is related to another by the bisimulation relation if they agree on their common atomic propositions, and in addition, for every successor of one state there is a corresponding successor of the other state, *and vice versa*. Formally, we have the following.

Definition 8 Let $M_1 = (AP_1, S_1, I_1, R_1, [\![\cdot]\!]_1)$ and $M_2 = (AP_2, S_2, I_2, R_2, [\![\cdot]\!]_2)$ be Kripke structures. A relation $B \subseteq S_1 \times S_2$ is a *bisimulation relation [83] between* M_1 *and* M_2 if for every $s_1 \in S_1$ and $s_2 \in S_2$ such that $B(s_1, s_2)$, the following conditions hold:

- For all $p \in AP_1 \cap AP_2$, $[\![p]\!]_1(s_1)$ iff $[\![p]\!]_2(s_2)$,
- $\forall\, t_1\ [[\ R_1(s_1, t_1)] \Rightarrow \exists\, t_2\ [[\ R_2(s_2, t_2) \wedge B(t_1, t_2)\ ]]]$, and
- $\forall\, t_2\ [[\ R_2(s_2, t_2)] \Rightarrow \exists\, t_1\ [[\ R_1(s_1, t_1) \wedge B(t_1, t_2)\ ]]]$

We write $s_1 \equiv s_2$ for $B(s_1, s_2)$. We say that M_1 *is bisimilar to* M_2 (denoted $M_1 \equiv M_2$) if there exists a bisimulation relation B between M_1 and M_2 such that

- $\forall\, s_1 \in I_1\ [\exists\, s_2 \in I_2\ [B(s_1, s_2)]]$ and
- $\forall\, s_2 \in I_2\ [\exists\, s_1 \in I_1\ [B(s_1, s_2)]]$

Example 3 Consider the Kripke structure $M_5 = (AP_5, S_5, I_5, R_5, [\![\cdot]\!]_5)$ in Fig. 4, where $AP_5 = \{F_0\}$, $S_5 = \{a'_1, a'_2, a'_3, a'_4, a'_5\}$, and $I_5 = \{a'_1\}$. The transition relation and the state labeling are shown in the figure, where, as before, state labeling includes negation of atomic propositions explicitly.

Also consider the Kripke structure M of Fig. 2. In order to show that $M \equiv M_5$ we first notice that $AP_5 \subseteq AP$. A simulation relation $B \subseteq S \times S_5$ between M and M_5 can be defined as follows:

$$B = \{(s_1, a'_1), (s_2, a'_2), (s_3, a'_2), (s_4, a'_4), (s_5, a'_3), (s_6, a'_4), (s_7, a'_5), (s_8, a'_5)\}.$$

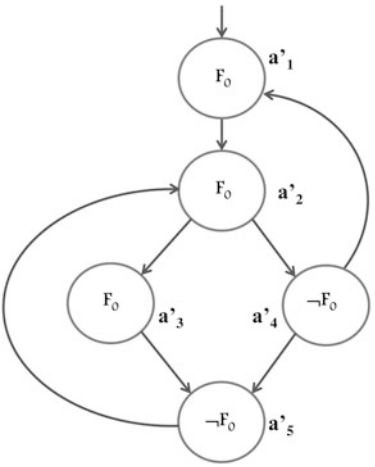


Fig. 4 Kripke structure M_5, bisimilar to Kripke structure M of the mutual exclusion program

It is easy to check that corresponding states agree on the common atomic propositions, and that for every successor of one state there is a corresponding successor of the other state, and vice versa.

For example, $(s_2, a'_2) \in B$ and for the successors s_4, s_5 of s_2 it holds that $(s_4, a'_4) \in B$ and $(s_5, a'_3) \in B$, where a'_4, a'_3 are successors of a'_2. Moreover, for the successors a'_3 and a'_4 of a'_2 it holds that $(s_4, a'_4) \in B$ and $(s_5, a'_3) \in B$, where s_4, s_5 are successors of s_2.

Note that if we extend AP_5 with additional atomic propositions from M, then the smallest structure that is bisimilar to M is M itself. To see why this is true, consider for instance $AP'_5 = \{T_1, F_0\}$. Now s_2 and s_3 cannot both be related to a'_2, since they do not agree on their labeling. As a result, s_7 and s_8 will have to be related to different abstract states, since they do not agree on their successors, and similarly for s_4 and s_6. This demonstrates the trade-off between the set of properties that can be checked in a reduced structure and the size (in terms of states and transitions) of this model. □

Similarly to the simulation relation, the bisimulation relation induces a relationship between paths in M_1 and M_2, as described in the following lemma.

Lemma 2 ([15]) *Let B be a bisimulation relation between M_1 and M_2 and assume that $s_1 \in S_1$, $s_2 \in S_2$, and $B(s_1, s_2)$. Then, for every maximal path π_1 from s_1 there is a maximal path π_2 from s_2 such that $length(\pi_1) = length(\pi_2)$. Further, for every $i \leq length(\pi_1)$, $B(\pi_1[i], \pi_2[i])$.*

The following theorem states the preservation of formulas in the μ-calculus and in CTL* and CTL.

Theorem 2 *Let $M_1 \equiv M_2$, $s_1 \equiv s_2$, and let φ be a formula with atomic propositions in $AP_1 \cap AP_2$. Then the following holds*:

1. [76] *Assume that* φ *is in* $\mathscr{L}_\mu$. *Then* $[\![\varphi]\!]_1(s_1)$ *if and only if* $[\![\varphi]\!]_2(s_2)$ *and* $[\![\varphi]\!]_1(M_1)$ *if and only if* $[\![\varphi]\!]_2(M_2)$.
2. [15] *Assume that* φ *is in CTL*. Then* $[\![\varphi]\!]_1(s_1)$ *if and only if* $[\![\varphi]\!]_2(s_2)$ *and* $[\![\varphi]\!]_1(M_1)$ *if and only if* $[\![\varphi]\!]_2(M_2)$.

As before, the result holds for CTL and LTL formulas as well, as sublogics of CTL*.

Consider Example 3 again. Consider also the CTL* property $\varphi = \forall \mathsf{GF}(\exists \mathsf{X} F_0 \wedge \exists \mathsf{X} \neg F_0)$. The formula means that along every path, infinitely often we reach a state which has two successors, one that satisfies F_0 and the other that satisfies $\neg F_0$. In M_2, a_2' and a_4' are such states. In M, it is states s_2, s_3, s_4 and s_6.

Since $M_2 \models \varphi$, by Theorem 2 we conclude that $M \models \varphi$ as well. Note that this formula is not preserved by the simulation relation since it is not a universal formula.

Next, we describe an algorithm [15] to compute bisimulation equivalence over a single structure. Let $M = (AP, S, I, R, [\![\cdot]\!])$ be a finite Kripke structure. We compute a sequence of equivalence relations over S. We start by relating two states if and only if they agree on all atomic propositions. We iteratively refine the relation until a fixpoint is reached. Upon termination, two states are related if and only if they are bisimilar.

- $E_0(s, s')$ if and only if $L(s) = L(s')$.
- $E_{n+1}(s, s')$ if and only if
  - $E_n(s, s')$ and
  - $\forall\, s_1\ [[\ R(s, s_1)] \Rightarrow \exists\, s_1'\ [[\ R(s', s_1') \wedge E_n(s_1, s_1')\ ]]]$ and
  - $\forall\, s_1'\ [[\ R(s', s_1')] \Rightarrow \exists\, s_1\ [[\ R(s, s_1) \wedge E_n(s_1, s_1')\ ]]]$

It is easy to see that when $E_n = E_{n+1}$, E_n is a bisimulation relation. Further, it is the greatest bisimulation over M. We denote by E the resulting relation. The relation E induces the following set of equivalence classes: For each $s \in S$, $[s] = \{s' \in S \mid (s, s') \in E\}$. We can now define a *quotient structure* $M_A = (AP, S_A, I_A, R_A, [\![\cdot]\!]_A)$, that is the smallest structure, in terms of states and transitions, which is bisimulation equivalent to M:

- $S_A = \{\ [s] \mid s \in S\ \}$
- $I_A = \{\ [s_0] \mid s_0 \in I\ \}$
- $R_A = \{\ ([s], [t]) \mid \exists s_1, t_1 \in S\ [(s_1, t_1) \in R \wedge s_1 \in [s] \wedge t_1 \in [t]\ ]\}$
- for every $p \in AP$: $[\![p]\!]_A([s]) = [\![p]\!](s)$

13.3.3 Additional Reading

Many notions of equivalence relations over models and their related logic preservation have been defined. For example, [43, 44, 56, 96].

An efficient algorithm for computing the quotient structure with respect to bisimulation is suggested in [74]. Other symbolic algorithms for bisimulation minimization are proposed in [12, 13]. A notion of *simulation equivalence* and its related

quotient structure was introduced in [20]. An efficient algorithm for computing the quotient structure with respect to simulation is presented in [57].

13.4 Abstraction Based on Simulation

13.4.1 Existential Abstraction

In this section we define an abstraction which is suitable for reasoning about universal properties, such as all properties expressible in $\Box\mathscr{L}_\mu$. We define an abstract model (Kripke structure), which is derived from a given concrete model by means of a concretization function γ. The abstract model is, by definition, guaranteed to be greater by the simulation relation than the concrete model, thus preservation of universal properties is guaranteed by Theorem 1.

It is important to note that while the definition assumes that a concrete model is given, in practice such a model is usually too large to fit into memory and therefore is not produced. Abstract models are constructed directly from some high-level description of the system [31, 34]. More details on the construction of abstract models in practice can be found elsewhere in this Handbook [65] (Chap. 15). Construction of abstractions is also the topic of *Abstract Interpretation* [8, 37–40, 76].

We define abstract Kripke structures by means of *existential abstraction* [34]. Let M be a Kripke structure over a set S of states. Given a set $\widehat{S}$ of abstract states, the *concretization function* $\gamma : \widehat{S} \to 2^S$ indicates, for each abstract state $\widehat{s}$, the set of concrete states represented by $\widehat{s}$. Existential abstraction defines an abstract state to be an initial state if it represents an initial concrete state. Similarly, there is a transition from abstract state $\widehat{s}$ to abstract state $\widehat{s}'$ if there is a transition from a state represented by $\widehat{s}$ to a state represented by $\widehat{s}'$. In order to preserve the valuation of atomic propositions in the abstract model we will use a γ that is *appropriate* for AP. That is, for every $p \in AP$ and every s_1, s_2 and $\widehat{s}$ such that $s_1, s_2 \in \gamma(\widehat{s})$, we have $p(s_1) \Leftrightarrow p(s_2)$. In Sect. 13.6 we will investigate abstract models for which γ is not appropriate.

An abstract model constructed by means of an existential abstraction is an *over-approximation* of the concrete model in the sense that every behavior of the concrete model corresponds to a prefix of a behavior in the abstract model, but the abstract model may also contain additional behaviors. Formally,

Definition 9 Let $M = (AP, S, I, R, [\![\cdot]\!])$ be a Kripke structure, let $\widehat{S}$ be a set of states and $\gamma : \widehat{S} \to 2^S$ be a concretization function. The *Kripke structure* $\widehat{M} = (AP, \widehat{S}, \widehat{I}, \widehat{R}, \widehat{[\![\cdot]\!]})$ *derived from* M *by* γ is defined as follows:

1. $\widehat{I}(\widehat{s}) \Leftrightarrow \exists s\,(s \in \gamma(\widehat{s}) \wedge I(s))$.
2. $\widehat{R}(\widehat{s_1}, \widehat{s_2}) \Leftrightarrow \exists s_1 \exists s_2\,(s_1 \in \gamma(\widehat{s_1}) \wedge s_2 \in \gamma(\widehat{s_2}) \wedge R(s_1, s_2))$.
3. For all $p \in AP$, $\widehat{[\![p]\!]}(\widehat{s}) \Leftrightarrow \forall_{s \in \gamma(\widehat{s})} [\![p]\!](s)$.

In this context, we refer to $\widehat{M}$ as the *abstract Kripke structure*, to its states as *abstract states*, etc., and to M as the concrete Kripke structure.

Note the "iff" ($\Leftrightarrow$) conditions in each of the items. $\widehat{M}$ defined this way is called the *exact* abstraction derived from M by γ. If the first two "iff"s are replaced by "if" ($\Leftarrow$) then more initial states and more transitions are possible in the abstract model. It is then referred to as an *approximated* abstraction.

As for the third item, the "iff" condition is necessary in order to guarantee that atomic propositions true in an abstract state $\widehat{s}$ are also true in all concrete states in $\gamma(\widehat{s})$. Falsity of atomic propositions is preserved only if γ is appropriate. In that case, $p(\widehat{s}) \Leftrightarrow p(s)$ for every $p \in AP$ and $s \in \gamma(\widehat{s})$.

Note that the abstract model $\widehat{M}$ defined above is just a Kripke structure. Usual model-checking algorithms can be applied to it. However, only if a universal property is true in $\widehat{M}$ can we conclude that it is also true in the model M from which $\widehat{M}$ has been derived.

The following theorem and corollary state that universal properties which are correct for $\widehat{M}$ are correct for M as well.

Theorem 3 ([34, 40, 76]) *Let M be a Kripke structure defined over AP. Further, let γ be appropriate for AP and $\widehat{M}$ be an abstract model derived from M by γ. Then $M \leq \widehat{M}$.*

Proof Sketch We define the following relation over S and $\widehat{S}$: For every $s \in S$ and $\widehat{s} \in \widehat{S}$, $H(s, \widehat{s})$ if and only if $s \in \gamma(\widehat{s})$. It is easy to see that H is a simulation relation from M to $\widehat{M}$. Thus, we conclude that $M \leq \widehat{M}$.

By the above theorem and Theorem 1 we get the following.

Corollary 1 *Let M be a Kripke structure and φ be a universal formula, both defined over AP. Further, let γ be appropriate for AP and $\widehat{M}$ be an abstract model derived from M by γ. Then $[\![\varphi]\!](\widehat{M})$ implies $[\![\varphi]\!](M)$.*

Note that, for exact abstractions, once $\widehat{S}$, γ, and AP are given, $\widehat{I}$ and $\widehat{R}$ are uniquely determined. Thus, $\widehat{S}$, γ, and AP uniquely determine $\widehat{M}$.

Example 4 Consider again the Kripke structure M of Fig. 2. An abstraction of M can be defined by choosing $AP_1 = \{C_1, C_2, F_0\}$, $\widehat{S_1} = \{a_1, a_2, a_3\}$, where $\gamma_1(a_1) = \{s_1, s_2, s_3, s_5\}$, $\gamma_1(a_2) = \{s_4, s_7\}$, and $\gamma_1(a_3) = \{s_6, s_8\}$. The resulting abstract Kripke structure M_1 is shown in Fig. 3. In Example 2 we gave a simulation relation from M to M_1. By Theorem 3 we now know that such a simulation relation exists, by the way M_1 is constructed.

By Item 2 of Theorem 1, since the transition relation of M is total and since $M_1 \models \forall\mathsf{G}\neg(C_1 \wedge C_2)$ we conclude that $M \models \forall\mathsf{G}\neg(C_1 \wedge C_2)$ as well. □

Example 5 The abstraction above is suitable for proving the mutual exclusion property for M. In order to prove another property, for instance a property of process P_1 alone, we may need to use a different abstraction. Let $AP_2 = \{N_1, T_1, C_1\}$, $\widehat{S_2} =$

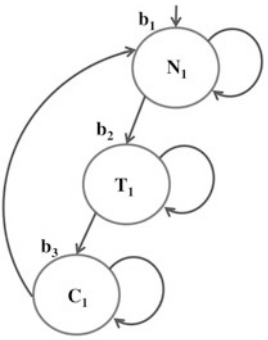


Fig. 5 The abstract Kripke structure M_2

$\{b_1, b_2, b_3\}$, and let $\gamma_2(b_1) = \{s_1, s_3, s_6\}$, $\gamma_2(b_2) = \{s_2, s_5, s_8\}$, and $\gamma_2(b_3) = \{s_4, s_7\}$. The resulting abstract Kripke structure M_2 is given in Fig. 5.

Consider the property $\forall\mathsf{G}(N_1 \rightarrow \forall(\neg C_1 \mathsf{W} T_1))$, where W is the temporal operator expressing "weak until." It specifies that every path starting in the neutral state and reaching the critical state must pass through the trying state. Since $M_2 \models \forall\mathsf{G}(N_1 \rightarrow \forall(\neg C_1 \mathsf{W} T_1))$, we conclude that the property holds for M as well.

On the other hand, we cannot prove non-starvation of P_1 using the abstraction M_2. That is, we cannot prove $M_2 \models \forall\mathsf{G}(T_1 \rightarrow \forall\mathsf{F}C_1)$. In this case, model checking will return an abstract counterexample of the form $b_1^+(b_2)^\omega$. Our next goal is then to determine whether there is a corresponding counterexample in M or whether the abstract counterexample is *spurious*. In the latter case, we will apply *refinement* in order to eliminate the spurious counterexample from the abstract model. The refinement will result in an (abstract) model which is "closer" in some sense to M. We discuss this topic in more detail in Sect. 13.5.

An alternative approach to checking whether non-starvation holds for P_1 would be to check the negation of the property, i.e., $\exists\mathsf{F}(T_1 \wedge \exists\mathsf{G}\neg C_1)$, on M_2. We will find out that indeed $M_2 \models \exists\mathsf{F}(T_1 \wedge \exists\mathsf{G}\neg C_1)$. However, the existential abstraction provides no soundness guarantee for formulas that are not universal. Thus, we cannot conclude potential starvation of P_1 in the original model M. In Sect. 13.6 we remedy this problem by suggesting abstractions which are sound for the full μ-calculus.

13.4.1.1 Specific Abstractions: Examples

Several types of abstractions based on existential abstraction are defined and used. The most commonly used are *localization reduction* [70], which is mostly used for hardware verification (e.g., [6]), and *predicate abstraction* [47], which is more suitable for software verification. The abstractions differ in their choice of abstract states and the concretization function.

For a hardware design, localization reduction distinguishes between visible and invisible Boolean variables, where only the visible variables are considered to be relevant for the checked property. The abstract states are chosen to be all valuations of the visible variables. γ maps each abstract state to the set of concrete states that

agree with it on the valuation of the visible variables (while they may differ in their valuation of the invisible variables). Usually, the visible variables are also chosen as the set AP of atomic propositions.

Predicate abstraction chooses a set of predicates over the program variables. Abstract states correspond to possible valuations of these predicates. An abstract state represents all those concrete states which agree with it on the valuation of the predicates. The predicates are also used as the atomic propositions in AP.

As mentioned before, once M, $\widehat{S}$, γ, and AP are chosen, the exact abstract model for M, defined by means of existential abstraction, is uniquely determined. Approximated abstract models for M can be defined as well by allowing more initial states and transitions.

Other types of abstractions can also be found in the literature, e.g., [34]. A more detailed description of the abstractions mentioned above, including methods for computing abstract models, and tools that implement them can be found in [48] and elsewhere in this Handbook [65] (Chap. 15).

13.4.2 Additional Reading

Extensions, improvements, and applications of predicate abstraction in the context of software verification are widely investigated [3, 5, 7, 29, 33, 36]. Predicate abstraction is also applied in hardware verification [63, 68], and in the verification of sequential [77, 78] and concurrent [102] Linux device drivers.

An important question is how to compute the needed predicates. This can be done, for instance, using theorem provers [86, 87], symbolic decision procedures [71], interpolation [64], and interpolation sequences [58].

Some work aims to minimize the increase in size of abstract models due to refinement. Lazy abstraction, for instance, adds new predicates to the model only when needed and where needed [59, 60, 69, 78, 99, 100].

13.5 CounterExample-Guided Abstraction Refinement (CEGAR)

Regardless of the type of abstraction we use, the abstract model $\widehat{M}$ generally contains less information than the concrete model M. Thus, model checking the structure $\widehat{M}$ potentially produces incorrect results. Corollary 1 guarantees that if a universal property is true in $\widehat{M}$ then it is also true in M. On the other hand, the following example shows that if the abstract model invalidates an $\Box\mathscr{L}_\mu$ specification, *the actual model may still satisfy the specification*.

Example 6 The traffic light controller M, presented on the left-hand side of Fig. 6, contains three states, *red*, *green*, and *yellow*. We define $\{ISred\}$ to be the set of

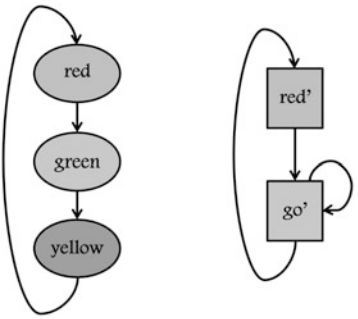


Fig. 6 Abstraction of a traffic light

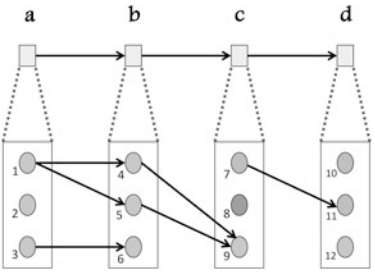


Fig. 7 Spurious counterexample. The abstract state c is a failure state

atomic propositions, where

$$ISred(red) = true \quad \text{and} \quad ISred(green) = ISred(yellow) = false.$$

We would like to prove for M the property "along every path, infinitely often ($ISred$) holds." This can be written in CTL as $\psi = \forall \mathsf{G} \forall \mathsf{F}(ISred)$. We choose the set of abstract states to be $\{red', go'\}$, where $\gamma(red') = \{red\}$ and $\gamma(go') = \{green, yellow\}$. Clearly,

$$ISred\big(red'\big) = true \quad \text{and} \quad ISred\big(go'\big) = false.$$

It is easy to see that $M \models \psi$ while $\widehat{M} \not\models \psi$. There exists an infinite abstract path $red', go', go', \ldots$ that invalidates the property. However no corresponding concrete trace exists in M. □

When an abstract counterexample does not correspond to any concrete counterexample, we call it *spurious*. For example, $red', go', go', \ldots$ in the above example is a spurious counterexample.

Let us consider the situation outlined in Fig. 7. There, a, b, c, and d are abstract states, where $\gamma(a) = \{1, 2, 3\}$, $\gamma(b) = \{4, 5, 6\}$, $\gamma(c) = \{7, 8, 9\}$, and $\gamma(d) = \{10, 11, 12\}$. The arrows between concrete states describe concrete transitions where the arrows between abstract states describe abstract transitions, defined according to Definition 9. States 1, 2, 3 are initial states of the concrete model and a is the initial state of the abstract model.

We see that the abstract path a, b, c, d does not have a corresponding concrete path. Every concrete path that starts at the initial state and that reaches a state in

$\gamma(c)$ ends up in state 9, from which we cannot go further. Therefore, 9 is called a *dead-end state*. On the other hand, state 7 is a *bad state*, since it made us believe that there is an outgoing transition. Finally, state 8 is an *irrelevant state* since it is neither dead-end nor bad. To eliminate the spurious path, the abstraction should be refined in a way that separates dead-end states from bad states. These notions are made precise in the next section.

13.5.1 The CEGAR Framework

In this section we present the framework of *CounterExample-Guided Abstraction-Refinement* (CEGAR) [30, 35, 70], for universal temporal logics and existential abstraction. The framework is suitable, in principle, for logics such as $\Box\mathscr{L}_\mu$, ACTL*, ACTL, and LTL. In practice, however, most model-checking tools handle CTL or LTL. They usually produce a counterexample in the form of a finite path leading to a state violating the property. Alternatively, they produce a counterexample in the form of a lasso (a finite path leading to a simple cycle), showing a behavior along which a desired state is never reached. Most CEGAR implementations refer to these forms of counterexamples.

The main steps of the CEGAR framework are presented below:

1. Given a system $\mathscr{P}$ (whose concrete model is M) and a universal temporal formula φ, generate an initial abstract model $\widehat{M}$.

   This step is typically done by examining a high-level description of $\mathscr{P}$. For software, for instance, we may examine the program text and choose conditions used in control statements such as **if** and **while** as predicates. Additional predicates come from the atomic formulas in φ. Proceed to step 2.
2. Model check $\widehat{M}$ with respect to φ. If $\widehat{M}$ satisfies φ, then conclude that the concrete model M satisfies the formula and stop. If a counterexample $\widehat{T}$ is found, check whether it is also a counterexample in the concrete model. If it is, conclude that the concrete model does not satisfy the formula and stop. Otherwise, the counterexample is spurious. Proceed to step 3.
3. Refine the abstract model, so that $\widehat{T}$ will not be included in the new, refined abstract model. Go back to step 2.

   Refinement is typically done by *partitioning* an abstract state. By this we mean that the set of concrete states represented by the abstract state, is partitioned. The refinement can be accelerated at the cost of faster growth of the abstract model if the criterion obtained for partitioning one abstract state (e.g., a new predicate) is used to partition all abstract states. By accelerating we mean that fewer refinement iterations will be needed before CEGAR terminates with either a verification of the checked property or with a concrete counterexample.

13.5.1.1 Identifying Spurious Path Counterexamples

Assume the counterexample $\widehat{T}$ is a path (not necessarily maximal) $\widehat{s_1}, \ldots, \widehat{s_n}$ starting at the initial abstract state $\widehat{s_1}$. We extend the concretization function γ to sequences

of abstract states in the following way: $\gamma(\widehat{T})$ is the set of concrete paths defined as follows:

$$\gamma(\widehat{T}) = \left\{ \langle s_1, \ldots, s_n \rangle \;\middle|\; \bigwedge_{i=1}^{n} s_i \in \gamma(\widehat{s_i}) \wedge I(s_1) \wedge \bigwedge_{i=1}^{n-1} R(s_i, s_{i+1}) \right\}.$$

Next, we describe an algorithm to compute a sequence of sets of states that correspond to $\gamma(\widehat{T})$. Let $S_1 = \gamma(\widehat{s_1}) \cap I$. For $1 < i \leq n$, we define S_i in the following manner: $S_i := Image(S_{i-1}, R) \cap \gamma(\widehat{s_i})$, where $Image(S_{i-1}, R)$ is the set of successors, in M, of states in S_{i-1}. The sequence of sets S_i can be computed, for instance, using BDDs and the standard image computation algorithm.

The following lemma explains how to use the sets of states computed by the above algorithm in order to determine whether $\widehat{T}$ is spurious.

Lemma 3 ([30]) *The following are equivalent*:

1. *The path* $\widehat{T}$ *corresponds to a concrete counterexample.*
2. *The set* $\gamma(\widehat{T})$ *of concrete paths is non-empty.*
3. *For all* $1 \leq i \leq n$, $S_i \neq \emptyset$.

Suppose that condition (3) of Lemma 3 is violated, and let i be the largest index such that $S_i \neq \emptyset$. Then $\widehat{s_i}$ is called the *failure state* of the spurious counterexample $\widehat{T}$. Since $S_{i+1} = \emptyset$, S_i is the set of dead-end states which are reachable in m from an initial state but cannot go any further. $preImage(S_{i+1}, R)$ includes all predecessors of states in S_{i+1}. These are the bad states. In order to eliminate $\widehat{T}$, refinement should split $\gamma(\widehat{S_i})$ by separating dead-end states from bad states. It follows from Lemma 3 that if $\gamma(\widehat{T})$ is empty (i.e., if the counterexample $\widehat{T}$ is spurious), then there exists a minimal i $(1 \leq i \leq n)$ such that $S_i = \emptyset$.

Example 7 Consider a program with only one variable over domain $D = \{1, \ldots, 12\}$. Assume that the concretization function is shown in Fig. 7 by the dotted lines from $\widehat{S} = \{a, b, c, d\}$. Suppose that we obtain an abstract counterexample $\widehat{T} = a, b, c, d$. It is easy to see that $\widehat{T}$ is spurious. Using the terminology of Lemma 3, we have $S_1 = \{1, 2, 3\}$, $S_2 = \{4, 5, 6\}$, $S_3 = \{9\}$, and $S_4 = \emptyset$. Since S_4 is empty, c is the failure state. □

It should be noted that checking whether a counterexample is spurious and then finding a splitting criterion involves computations on the concrete model. These computations, however, are usually easier than applying model checking to the concrete structure. This is because they refer to the part of it which is relevant to a particular counterexample. BDD-based (e.g., [4, 30]) and SAT-based (e.g., [23, 62, 75]) CEGAR are typically used when the concrete model M is finite and moderately large, while SAT-based methods can usually handle larger models. If the concrete model is infinite-state then CEGAR usually applies SAT Modulo Theory (SMT)

or theorem proving (e.g., [5, 60]) in order to determine whether the counterexample is spurious and in order to find a suitable refinement. In some works, though, SAT-based CEGAR is applied to infinite-state concrete models as well [33, 36].

The refinement procedure refines the abstraction mapping by partitioning abstract states (that is, the sets of concrete states represented by them). Refinement is applied iteratively until a real counterexample is found, or the property is verified. If the concrete model is finite, then the refinement procedure is guaranteed to terminate.

13.5.2 Additional Reading

Depending on the type of γ and the size of M, and whether M is finite or infinite, the initial abstract model (i.e., abstract initial states and abstract transitions) can be built using BDDs, SAT solvers or theorem provers. Similarly, the partitioning of abstract states, performed in the refinement, can be done using BDDs (e.g., as in [30] and [4]), SAT solvers (e.g., as in [23, 62, 75]), linear programming and machine learning (e.g., as in [32]) or theorem provers (e.g., as in [5]).

While the focus in many papers is on counterexamples that are *finite paths*, [30] also handles counterexamples consisting of a finite path followed by a loop (lasso). In [35] CEGAR for all of ACTL is handled, with tree-like counterexamples.

There are many extensions, improvements, and applications of CEGAR, (e.g., [89]). The CEGAR framework is combined, for instance, with Bounded Model Checking (BMC) [53, 54] and with learning assumption for compositional reasoning [11, 22, 95].

An iterative abstraction-based verification method which does not use counterexamples is presented in [79]; a combination of the latter with CEGAR is presented in [2].

In [9] a technique is proposed for adjusting the level of abstraction during the analysis, for example, in combination analyses, to make one abstract domain more fine-grained and another more coarse.

Several tools have been developed based on predicate abstraction and on the abstraction-refinement framework. They are successfully used both in academic research and in practice, on industrial examples. A partial list includes SLAM [5], BLAST [7], SATABS [33], KRATOS [28], and Wolverine [69].

13.6 Abstraction Based on Modal Simulation

The simulation-based abstraction framework from the previous sections is sound for the universal fragment $\Box\mathscr{L}_\mu$ of the μ-calculus.[5] As shown in Example 5, soundness

[5] We focus on $\mathscr{L}_\mu$ in this section, but the results can be extended to CTL*-like logics.

breaks when non-universal formulas (containing the $\Diamond$ operator) are considered. Hence, a natural question is how to restore soundness in the presence of $\Diamond$s, i.e., soundness for the full μ-calculus. This problem has been investigated in several papers, e.g., [40, 46, 49, 61]. One way of summarizing their conclusions is as follows: When abstracting a Kripke structure in the presence of both $\Box$ and $\Diamond$ in the property logic, the abstraction needs to have two transition relations, i.e., two types of transitions between states: one over which to interpret $\Box$, and another for $\Diamond$. These different types of transition are usually called *may* and *must*, respectively. They need to obey the following requirements:

1. For every concrete transition (i.e., transition in the concrete Kripke structure), there needs to be a corresponding *may* transition in the abstract structure.
2. For every *must* transition in the abstraction, there needs to be a corresponding concrete transition.

The meaning of the word "corresponding" in these requirements is formalized in Definition 11 below. For now, suffice it to say that it is just like in the definition of simulation. Note that from the two requirements above, it follows that when there is a *must* transition between two states, then there is also a *may* transition between those two states; in other words, the *must* transition relation is a subset of the *may* transition relation. Kripke structures with *may* and *must* transitions are called Kripke Modal Transition Systems (KMTSs) [46, 61]. KMTSs differ from Kripke structures in another aspect. Their propositional interpretation $[\![\cdot]\!]$ (or labeling function) is defined with respect to the set *Lit* of literals, which includes the propositions from *AP* and their negations. In the abstraction framework presented in this section, KMTSs play the role of the abstract objects.

Definition 10 A *Kripke Modal Transition System (KMTS) over AP* is a tuple $M = (AP, S, I, R_{\text{must}}, R_{\text{may}}, [\![\cdot]\!])$, where S is a set of states, $I \subseteq S$ is the set of initial states, $R_{\text{must}} \subseteq S \times S$ and $R_{\text{may}} \subseteq S \times S$ are transition relations such that $R_{\text{must}} \subseteq R_{\text{may}}$, and $[\![\cdot]\!]$ turns literals in *Lit* into state predicates, i.e., for $p \in AP$ and $s \in S$, $[\![p]\!](s)$ and $[\![\neg p]\!](s)$ are values in $\{\text{tt}, \text{ff}\}$, with the requirement that they cannot both be tt.

Example 8 Figure 8 describes the KMTS M_3, where dotted lines represent the *may* transitions in R_{may} and solid lines represent the *must* transitions in R_{must}. States are labeled with all the literals that are true in those states. For example, $[\![F_0]\!](b_2) = \text{tt}$ and $[\![\neg F_0]\!](b_4) = \text{tt}$. However, $[\![F_0]\!](b_3)$ and $[\![\neg F_0]\!](b_3)$ are both false, meaning that the value of F_0 in state b_3 is unknown (or undefined).

Consider again the Kripke structure M_1 of Fig. 3. In Example 4, M_1 is presented as an abstract Kripke structure, obtained by existential abstraction from the Kripke structure M in Fig. 2. M_1 can also be viewed as a KMTS abstracting M by setting $R_{\text{may}} = R_1$ and $R_{\text{must}} = \emptyset$. In addition, $[\![\cdot]\!]_1$ is extended to negated propositions by setting $[\![\neg p]\!]_1(s) = \text{tt}$ if and only if $[\![p]\!]_1(s) = \text{ff}$. □

The concrete objects (the set C from Definition 1 of abstraction framework in Sect. 13.2.1) are Kripke structures throughout this chapter. However, the following

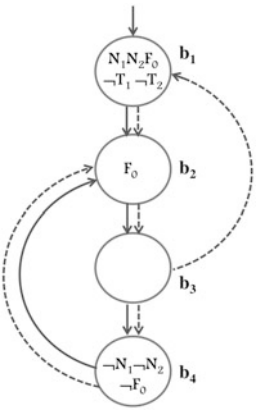


Fig. 8 A Kripke modal transition system (KMTS) M_3 abstracting the mutual exclusion program

definition of the abstraction relation based on modal simulation is not from Kripke structures to KMTSs as might be expected, but from KMTSs to KMTSs. This is a generalization, since a *concrete* Kripke structure can be seen as a KMTS in which the *may* and *must* transition relations coincide, i.e., $R_{\text{must}} = R_{\text{may}}$. The advantage of this generalization is that it allows us to take an abstract object and abstract it further, so that the process of abstraction can be broken up into steps. Indeed, the entire simulation-based framework can be recast as a special case of the framework based on modal simulation: A Kripke structure that *is used as an abstraction* can be seen as a KMTS in which all transitions are of type *may* and whose *must* transition relation is empty, as is done with Kripke structure M_1 in Example 8 above.

Definition 11 ([40, 46, 72]) Let $M_1 = (AP, S_1, I_1, R^1_{\text{must}}, R^1_{\text{may}}, [\![\cdot]\!]_1)$ and $M_2 = (AP, S_2, I_2, R^2_{\text{must}}, R^2_{\text{may}}, [\![\cdot]\!]_2)$ be KMTSs over AP and let $H \subseteq S_1 \times S_2$ be a relation. H is a *modal simulation from* M_1 *to* M_2 if and only if for every $s_1 \in S_1$ and $s_2 \in S_2$, we have that $H(s_1, s_2)$ implies:

1. for every $l \in Lit$, if $[\![l]\!]_2(s_2) = \text{tt}$ then $[\![l]\!]_1(s_1) = \text{tt}$; and
2. for every $s'_1 \in S_1$ such that $R^1_{\text{may}}(s_1, s'_1)$, there is some $s'_2 \in S_2$ such that $R^2_{\text{may}}(s_2, s'_2)$ and $H(s'_1, s'_2)$; and
3. for every $s'_2 \in S_2$ such that $R^2_{\text{must}}(s_2, s'_2)$, there is some $s'_1 \in S_1$ such that $R^1_{\text{must}}(s_1, s'_1)$ and $H(s'_1, s'_2)$.

If H is a modal simulation from M_1 to M_2 and furthermore $\forall s_1 \in I_1\ \exists s_2 \in I_2$: $(s_1, s_2) \in H$, then M_1 *is modal-simulated by* M_2, and M_2 *modal-simulates* M_1, denoted $M_1 \preceq M_2$. In this case we write $s_1 \preceq s_2$ for $H(s_1, s_2)$.

In the abstraction framework of this section, modal simulation between KMTSs ($\preceq$) plays the role of the abstraction relation (ρ) from Definition 1.

So far, we have defined the concrete and abstract objects, as well as the abstraction relation between them. Next, we define the evaluation of formulas from the full μ-calculus over KMTSs. As usual, this is done via a definition of how to evaluate a formula over a state of a KMTS.

Definition 12 Let $M = (AP, S, I, R_{\text{must}}, R_{\text{may}}, [\![\cdot]\!])$ be a KMTS. The definition of $[\![\cdot]\!]$ is extended as follows. For any formula $\psi \in \mathscr{L}_\mu$ and any $s \in S$, $[\![\psi]\!](s)$ is as in Definition 6, with the clauses for the modalities replaced by the following:

$$[\![\Box\psi]\!](s) = \begin{cases} \text{tt} & \text{if } \forall t \in S : R_{\text{may}}(s,t) \text{ implies } [\![\psi]\!](t) = \text{tt} \\ \text{ff} & \text{otherwise} \end{cases}$$

$$[\![\Diamond\psi]\!](s) = \begin{cases} \text{tt} & \text{if } \exists t \in S : R_{\text{must}}(s,t) \text{ and } [\![\psi]\!](t) = \text{tt} \\ \text{ff} & \text{otherwise} \end{cases}$$

The following theorem states a soundness result for abstraction based on modal simulation.

Theorem 4 *Let M_1 and M_2 be as in Definition 11, $s_1 \in S_1$, $s_2 \in S_2$, $\varphi \in \mathscr{L}_\mu$, and assume that $M_1 \preceq M_2$ and $s_1 \preceq s_2$. Then $[\![\varphi]\!](s_2) = tt$ implies $[\![\varphi]\!](s_1) = tt$ and $[\![\varphi]\!](M_2) = tt$ implies $[\![\varphi]\!](M_1) = tt$.*

Example 8 (Continued) We note that M of Fig. 2 is modal-simulated by M_3 of Fig. 8, by observing that the relation $H = \{(s_1, b_1), (s_2, b_2), (s_3, b_2), (s_4, b_3), (s_5, b_3), (s_6, b_3), (s_7, b_4), (s_8, b_4)\}$ is a modal simulation from M to M_3.

Consider the CTL formula $\varphi_3 = \exists\mathsf{G}\forall\mathsf{F}(F_0)$ which means that there exists a path along which it is always true that all its continuations eventually reach a state in which F_0 holds. By Theorem 4, since $[\![\varphi_3]\!](M_3) = \text{tt}$ we conclude that $[\![\varphi_3]\!](M) = \text{tt}$ as well.

13.6.1 A Three-Valued Setting

Recall that the notion of soundness that we have used so far (Definition 2 in Sect. 13.2.1) is called *soundness for true*, since negative results (i.e., finding that $[\![\varphi]\!]^\alpha(\widehat{M})$ is false when checking φ via some abstraction $\widehat{M}$ of a concrete Kripke structure M) do not carry over from the abstract to the concrete side. In the simulation-based framework from Sect. 13.4, when verification via an abstraction fails, another abstraction needs to be constructed, possibly by a refinement approach such as CEGAR, on which the verification attempt is then repeated. In the framework based on modal simulation, there is another thing that may be attempted when we find that $[\![\varphi]\!]^\alpha(\widehat{M})$ is false. Namely, we may try to verify $\neg\varphi$ via the same abstraction $\widehat{M}$. If we find that $[\![\neg\varphi]\!]^\alpha(\widehat{M})$ is true, then, by soundness for true, we may conclude that $[\![\neg\varphi]\!](M)$ is true, which is equivalent to $[\![\varphi]\!](M)$ being false. Note that this additional option is generally not possible in the simulation-based framework, because in that case the soundness result is for the universal fragment of the logic only; the negation of the formula will then not be universal, except when it does not contain modal operators, which is, arguably, not an interesting case.

Think of the abstract object $\widehat{M}$ as the set $\gamma(\widehat{M})$ of concrete objects that it abstracts. When verification of $[\![\varphi]\!]^\alpha(\widehat{M})$ returns true, then we know that $[\![\varphi]\!](M')$ is

true for all M' in $\gamma(\widehat{M})$. If, on the other hand, $[\![\varphi]\!]^\alpha(\widehat{M})$ returns false, then it can either be that $[\![\varphi]\!](M')$ is false for all M' in $\gamma(\widehat{M})$, or that $[\![\varphi]\!](M')$ is true for some and false for other objects M' in the set $\gamma(\widehat{M})$. Thus, getting *false* on the abstract side means "*false* or *don't know*" when carried over to the concrete side. The additional check that can be performed in the modal-simulation framework then serves as an attempt to separate *false* from *don't know*.[6]

Indeed, the definition of how to evaluate $\varphi \in \mathscr{L}_\mu$ on a KMTS (Definition 12) can be extended to include this additional check. More formally, it is altered so as to evaluate φ and $\neg\varphi$ together, and return the result as a truth value from $\{\text{tt}, \text{ff}, \bot\}$, with $\bot$ representing *don't know*.

In this 3-valued setting, some adjustments need to be made to Definition 1 of abstraction framework. In particular, $[\![\varphi]\!]$ can no longer be viewed as the set of objects for which φ is true; instead, it should be a function that maps objects to 3-valued truth values.

Definition 13 The *3-valued semantics* $[\![\varphi]\!]$ of $\varphi \in \mathscr{L}_\mu$ w.r.t. a KMTS M is a mapping from S to $\{\text{tt}, \text{ff}, \bot\}$ [16, 61]. The interesting cases in the definition are again those of the modalities.

$$[\![\Box\psi]\!](s) = \begin{cases} \text{tt} & \text{if } \forall t \in S : R_{\text{may}}(s,t) \text{ implies } [\![\psi]\!](t) = \text{tt} \\ \text{ff} & \text{if } \exists t \in S : R_{\text{must}}(s,t) \text{ and } [\![\psi]\!](t) = \text{ff} \\ \bot & \text{otherwise} \end{cases}$$

$$[\![\Diamond\psi]\!](s) = \begin{cases} \text{tt} & \text{if } \exists t \in S : R_{\text{must}}(s,t) \text{ and } [\![\psi]\!](t) = \text{tt} \\ \text{ff} & \text{if } \forall t \in S : R_{\text{may}}(s,t) \text{ implies } [\![\psi]\!](t) = \text{ff} \\ \bot & \text{otherwise} \end{cases}$$

A model-checking algorithm based on this adapted definition can be viewed as making better use of the modal information (the "mays and musts") that is available in a KMTS.

In particular, Definition 11 can be applied to a (concrete) Kripke structure M and an (abstract) KMTS $\widehat{M}$, by viewing the Kripke structure as a KMTS where $R_{\text{must}} = R_{\text{may}} = R$. For a Kripke structure, the 3-valued semantics then agrees with the concrete semantics. Thus, preservation of $\mathscr{L}_\mu$ formulas is guaranteed by the following theorem.

Theorem 5 ([46]) *Let M_1 and M_2 be as in Definition 11, $s_1 \in S_1$, $s_2 \in S_2$, $\varphi \in \mathscr{L}_\mu$, and assume that $s_1 \preceq s_2$. Then $[\![\varphi]\!](s_2) \neq \bot \Rightarrow [\![\varphi]\!](s_1) = [\![\varphi]\!](s_2)$.*

This theorem can be extended to be about whole KMTSs ("$[\![\varphi]\!](M_2) \neq \bot \Rightarrow \ldots$"). This is left as an exercise for the reader.

[6] Getting a *don't know* in this case is no guarantee that in the concretization there are elements which do and elements which do not satisfy the formula. It may still be the case that the formula is true in all concretizations, or that in all concretizations it is false. This depends on the *thoroughness* of the model-checking algorithm, among other things. See the additional reading in Sect. 13.6.3.

It is interesting to note that even though both bisimulation and modal simulation preserve full $\mathscr{L}_\mu$, modal simulation is more flexible and allows us to define several abstract models for a given concrete model and AP. Each of the abstractions allows to prove and refute a different subset of $\mathscr{L}_\mu$ (returning $\perp$ for the rest). For modal simulation there is a trade-off between the size of the model (in terms of the number of states) and the number of $\mathscr{L}_\mu$ formulas for which a definite answer can be obtained. Bisimulation, on the other hand, allows no such flexibility.

13.6.2 Refinement

We may now develop an iterative abstraction-refinement framework for KMTSs and specifications in the full μ-calculus. However, now the refinement does not involve spurious counterexamples since falsification obtained in the abstract model is guaranteed to hold also in the concrete model. Instead, the refinement is aimed at eliminating indefinite results from which no conclusion can be drawn on the concrete model. A CEGAR-like framework, called TVAR, is described in [49, 50, 93].

Example 9 Consider again the KMTS M_3 of Fig. 8. Let R_3 be its 3-valued transition relation where a *must* transition is evaluated to tt, a *may* transition to $\perp$ and no transition is evaluated to ff. According to this convention,

$$R_3(b_1, b_2) = R_3(b_2, b_3) = R_3(b_3, b_4) = R_3(b_4, b_2) = \text{tt}, \quad \text{and} \quad R_3(b_3, b_1) = \perp .$$

R_3 on any other pair of states is ff.

The property $\varphi = \forall \mathsf{G} \exists \mathsf{F}(N_1 \wedge N_2 \wedge F_0)$ means that the initial state is reachable from any other reachable state in the model. Checking $[\![\varphi]\!](M_3)$ results in the indefinite value $\perp$. A 3-valued model-checking algorithm such as [49] will return the abstract state b_3 as the state to be refined, where the transition $R_3(b_3, b_1) = \perp$ is the cause of the indefinite result.

Figure 9 demonstrates the KMTS M_4, obtained by splitting b_3 into states c_3 and c_4, where $\gamma(c_3) = \{s_5\}$ and $\gamma(c_4) = \{s_4, s_6\}$. After refinement, we get $[\![\varphi]\!](M_4) = \text{tt}$.

13.6.3 Additional Reading

Extensions of KMTS propose hyper-must [82, 90] and hyper-may transitions [94].

The idea of abstractions based on 3-valued semantics (sometimes extended to 4-valued) appeared for software verification in [55], and for hardware verification in the widely used methodology of (G)STE [19]. Refinement for this framework has been suggested in [26, 27, 52, 97]. The 3-valued semantics has been exploited to suggest a compositional framework in [92].

Multi-valued model checking, abstraction, and refinement have been investigated in [18, 24, 25, 80, 91].

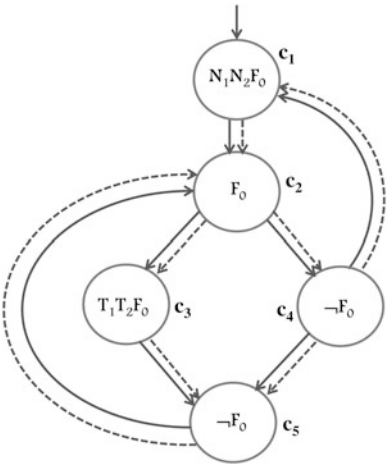


Fig. 9 The refined Kripke modal transition system M_4 for the mutual exclusion program

Other relevant works include [17, 46] (introducing the notion of *thorough semantics*) and [85, 103] (using 3-valued logic with shape analysis).

13.7 Completeness

This chapter has reviewed the use of Kripke Structures and Kripke Modal Transition Systems as abstractions of Kripke Structures. In the first case, simulation serves as the abstraction relation; in the second, modal simulation does. The move from using Kripke structures to using KMTSs as abstractions was motivated by the wish to extend the soundness result to include existential formulas, containing the $\exists$ path quantifier (in CTL*-like logics) or the $\Diamond$ modality (in the μ-calculus). In Example 8, it was suggested that KMTSs *generalize* Kripke structures as abstract objects: Any Kripke structure M that is used as an abstraction can be viewed as a KMTS in which all transitions of M are *may* transitions of the KMTS, and there are no *must* transitions. This reinterpretation of a Kripke structure (used as an abstraction) as a KMTS actually suffices to extend the soundness result to the full μ-calculus. Namely, by viewing the Kripke structure as a KMTS, we know how to interpret any formula in $\mathscr{L}_\mu$ over it (see Definition 12). It is just that, since there are no *must* transitions, any formula that contains a $\Diamond$ modality will evaluate to false on such a Kripke-structure-viewed-as-KMTS.

This observation leads to a reconsideration of what it is that we want from an abstraction. Do we want soundness for the full μ-calculus? We already have that with the reinterpretation provided above, even in the simulation-based framework. The unsatisfactory point is that this generalized soundness result for the simulation-based framework will still not help us to verify any formula that contains a $\Diamond$. Do we want more formulas with $\Diamond$s to evaluate to true on the abstraction? We can now have that: Once we are in the world of KMTSs, we can add *must* transitions to our abstraction to make it more informative (as long as it keeps modal-simulating the

concrete structure, of course). And the more *must* transitions we add to the abstraction, the more formulas with $\Diamond$s will evaluate to true on it.

Now that we do not have to worry about "extending the soundness to more types of formulas," the focus shifts to the question of which of the formulas that are true in a given concrete system can be verified via an abstraction, relative to an abstraction framework of choice. Clearly, the modal-simulation-based framework, introducing *must* transitions, extends the reach of verification-via-abstraction to certain existential formulas (as shown in the examples), whereas no existential formula could be verified in the simulation-based framework. Can *any* μ-calculus formula that is true on a given concrete structure be verified via a modal abstraction? If not, what else is needed, beyond *must* transitions? These questions motivate the introduction of the notion of *completeness*.

Definition 14 An abstraction framework is *complete* if and only if for every concrete object $c \in C$ and every property $\varphi \in L$ such that $[\![\varphi]\!](c) = v$ ($v \in \{\text{tt}, \bot, \text{ff}\}$), there exists a *finite* abstract object $a \in A$ such that $\rho(c, a)$ and $[\![\varphi]\!]^\alpha(a) = v$.

The requirement that the abstract object be finite may be interpreted somewhat loosely as saying that the evaluation of $[\![\varphi]\!]^\alpha(a)$ must be effectively computable. In practice, *efficient* computability will usually be required. Note that this definition takes into account the 3-valuedness that is inherent in working with abstractions. Furthermore, also on the concrete side, $[\![\varphi]\!](c)$ may evaluate to one of $\{\text{tt}, \bot, \text{ff}\}$; this is to accommodate composition of abstractions, in which the concrete objects are already abstractions of some form.

A review of the abstraction frameworks presented so far in the light of the notion of completeness is beyond the scope of this chapter. We refer to the bibliographic notes for a brief overview of the results in this area and for pointers to other notions of abstraction for Kripke structures that have been suggested as a result of the quest for completeness.

13.7.1 Additional Reading

In [82], a completeness result is given for the verification of branching-time properties of programs, modeled by Labeled Transition Systems (LTS), via abstractions. Abstraction is applied to the product of the program LTS and the property, expressed as an Alternating Transition System (ATS).

A notion of abstraction framework that is similar to the one used in this chapter, and the notion of completeness of such a framework, are introduced in [41] and [42]. In the former, several incompleteness results are given for branching-time logics. In particular, it is shown that a framework based on *reverse simulation*, which can be understood as using KMTSs with only *must* transitions, is incomplete for the *existential* fragment of branching-time logics, which is the dual of the universal fragment defined in this chapter. From this, it follows that the framework based on modal

simulation as presented in Sect. 13.6 is incomplete. This incompleteness is traced back to two shortcomings in KMTSs. These are then fixed by extending abstract objects with *focus operators* and *fairness*, resulting in an abstraction framework for branching-time properties, based on *Focused Transition Systems* and a game-based definition of the abstraction relation, that is shown to be complete. In [42], *tree automata* are suggested as an alternative complete abstraction framework.

The addition of focus operators to KMTSs is arguably equivalent to the generalization of *must* transitions to *must* hyper-transitions. This generalization is proposed in [73] in the context of bisimulation equation solving, extending the *Modal Transition Systems* of [72] to *Disjunctive Modal Transition Systems*. In [90], *must* hyper-transitions extend KMTSs to *Generalized KMTSs* with the goal of making the abstraction framework monotonic, meaning that refinement of an abstraction may cause formulas that are unknown in the original abstraction to become true or false in the refinement, but never the opposite (from a definite value to unknown).

The *abstract transition structures* introduced in [1] are also motivated by the need for *must* hyper-transitions, and they allow for a completeness result similar in spirit to that of [41].

In [45], completeness is studied in the context of another notion from abstraction theory, *expressiveness*. Under one of the considered definitions of expressiveness, KMTSs are shown to be strictly less expressive than Generalized KMTSs.

Whereas the generalization to *must* hyper-transitions is needed to achieve completeness specifically for the case of branching-time logics, the addition of fairness to abstractions, though equally necessary, is independent in that it arises in the case of linear-time logics as well [66, 98].

Acknowledgements We thank Yael Meller, Kedar Namjoshi, Tim Willemse, and the anonymous reviewers for their helpful comments.

The first author thanks (Alcatel-)Lucent Bell Laboratories, both in Murray Hill, NJ (USA) and in Antwerp (Belgium), where he was employed while working on this chapter.

References

1. de Alfaro, L., Godefroid, P., Jagadeesan, R.: Three-valued abstractions of games: uncertainty, but with precision. In: Symp. on Logic in Computer Science (LICS), pp. 170–179. ACM, New York (2004)
2. Amla, N., McMillan, K.L.: A hybrid of counterexample-based and proof-based abstraction. In: Hu, A.J., Martin, A.K. (eds.) Formal Methods in Computer Aided Design (FMCAD). LNCS, vol. 3312, pp. 260–274. Springer, Heidelberg (2004)
3. Ball, T., Podelski, A., Rajamani, S.K.: Boolean and Cartesian abstraction for model checking C programs. In: Margaria, T., Yi, W. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 2031, pp. 268–283. Springer, Heidelberg (2001)
4. Ball, T., Rajamani, S.: Boolean programs: a model and process for software analysis. Tech. Rep. 2000-14, Microsoft Research (2000)
5. Ball, T., Rajamani, S.K.: The SLAM toolkit. In: Berry, G., Comon, H., Finkel, A. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 2102, pp. 260–264. Springer, Heidelberg (2001)

6. Barner, S., Geist, D., Gringauze, A.: Symbolic localization reduction with reconstruction layering and backtracking. In: Intl. Conf. on Computer-Aided Verification (CAV). LNCS. Springer, Heidelberg (2002)
7. Beyer, D., Chlipala, A., Henzinger, T.A., Jhala, R., Majumdar, R.: The BLAST query language for software verification. In: Giacobazzi, R. (ed.) SAS. LNCS, vol. 3148, pp. 2–18. Springer, Heidelberg (2004)
8. Beyer, D., Henzinger, T.A., Théoduloz, G.: Configurable software verification: concretizing the convergence of model checking and program analysis. In: Damm, W., Hermanns, H. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 4590, pp. 504–518. Springer, Heidelberg (2007)
9. Beyer, D., Henzinger, T.A., Théoduloz, G.: Program analysis with dynamic precision adjustment. In: Intl. Conf. on Automated Software Engineering (ASE), pp. 29–38. IEEE, Piscataway (2008)
10. Biere, A., Kroening, D.: SAT-based model checking. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
11. Bobaru, M.G., Pasareanu, C.S., Giannakopoulou, D.: Automated assume-guarantee reasoning by abstraction refinement. In: Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 5123, pp. 135–148. Springer, Heidelberg (2008)
12. Bouajjani, A., Fernandez, J., Halbwachs, N.: Minimal model generation. In: Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 531, pp. 197–203. Springer, Heidelberg (1990)
13. Bouali, A., de Simone, R.: Symbolic bisimulation minimisation. In: Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 663, pp. 96–108. Springer, Heidelberg (1992)
14. Bradfield, J., Walukiewicz, I.: The mu-calculus and model checking. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
15. Browne, M.C., Clarke, E.M., Grumberg, O.: Characterizing finite Kripke structures in propositional temporal logic. Theor. Comput. Sci. **59**(1–2), 115–131 (1988)
16. Bruns, G., Godefroid, P.: Model checking partial state spaces with 3-valued temporal logics. In: Intl. Conf. on Computer-Aided Verification (CAV). LNCS, pp. 274–287. Springer, Heidelberg (1999)
17. Bruns, G., Godefroid, P.: Generalized model checking: reasoning about partial state spaces. In: Intl. Conf. on Concurrency Theory (CONCUR). LNCS, vol. 1877, pp. 168–182. Springer, Heidelberg (2000)
18. Bruns, G., Godefroid, P.: Model checking with multi-valued logics. In: Intl. Colloquium on Automata, Languages, and Programming (ICALP), pp. 281–293. Springer, Heidelberg (2004)
19. Bryant, R.E., Beatty, D.L., Seger, C.J.H.: Formal hardware verification by symbolic ternary trajectory evaluation. In: Design Automation Conf. (DAC), pp. 397–402 (1991)
20. Bustan, D., Grumberg, O.: Simulation-based minimization. ACM Trans. Comput. Log. **4**(2), 181–206 (2003)
21. Chaki, S., Gurfinkel, A.: BDD-based symbolic model checking. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
22. Chaki, S., Strichman, O.: Optimized L*-based assume-guarantee reasoning. In: Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, pp. 276–291. Springer, Heidelberg (2007)
23. Chauhan, P., Clarke, E., Kukula, J., Sapra, S., Veith, H., Wang, D.: Automated abstraction refinement for model checking large state spaces using SAT based conflict analysis. In: Formal Methods in Computer Aided Design (FMCAD). IEEE, Piscataway (2002)
24. Chechik, M., Devereux, B., Easterbrook, S., Gurfinkel, A.: Multi-valued symbolic model-checking. ACM Trans. Softw. Eng. Methodol. **12**, 371–408 (2003)
25. Chechik, M., Devereux, B., Gurfinkel, A., Easterbrook, S.: Multi-valued symbolic model-checking. Tech. Rep. CSRG-448, University of Toronto (2002)

26. Chen, Y., He, Y., Xie, F., Yang, J.: Automatic abstraction refinement for generalized symbolic trajectory evaluation. In: Formal Methods in Computer Aided Design (FMCAD). IEEE/ACM, Piscataway/New York (2007)
27. Chockler, H., Grumberg, O., Yadgar, A.: Efficient automatic STE refinement using responsibility. In: Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 4963. Springer, Heidelberg (2008)
28. Cimatti, A., Griggio, A., Micheli, A., Narasamdya, I., Roveri, M.: Kratos—a software model checker for SystemC. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV. LNCS, vol. 6806, pp. 310–316. Springer, Heidelberg (2011)
29. Cimatti, A., Narasamdya, I., Roveri, M.: Software model checking SystemC. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. **32**(5), 774–787 (2013)
30. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. J. ACM **50**(5), 752–794 (2003)
31. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
32. Clarke, E., Gupta, A., Kukula, J., Strichman, O.: SAT based abstraction-refinement using ILP and machine learning techniques. In: Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 2404. Springer, Heidelberg (2002)
33. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: SATABS: SAT-Based Predicate Abstraction for ANSI-C. In: Halbwachs, N., Zuck, L.D. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 3440, pp. 570–574. Springer, Heidelberg (2005)
34. Clarke, E.M., Grumberg, O., Long, D.E.: Model checking and abstraction. ACM Trans. Program. Lang. Syst. **16**(5), 1512–1542 (1994)
35. Clarke, E.M., Jha, S., Lu, Y., Veith, H.: Tree-like counterexamples in model checking. In: Symp. on Logic in Computer Science (LICS), pp. 19–29. IEEE, Piscataway (2002)
36. Clarke, E.M., Kroening, D., Sharygina, N., Yorav, K.: Predicate abstraction of ANSI-C programs using SAT. Form. Methods Syst. Des. **25**(2–3), 105–127 (2004)
37. Cousot, P.: Abstract interpretation. ACM Comput. Surv. **28**, 324–328 (1996)
38. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Symp. on Principles of Programming Languages (POPL), pp. 238–252. ACM, New York (1977)
39. Cousot, P., Cousot, R.: Temporal abstract interpretation. In: Symp. on Principles of Programming Languages (POPL), pp. 12–25. ACM, New York (2000)
40. Dams, D., Gerth, R., Grumberg, O.: Abstract interpretation of reactive systems. Trans. Program. Lang. Syst. **19**(2), 253–291 (1997)
41. Dams, D., Namjoshi, K.S.: The existence of finite abstractions for branching time model checking. In: Symp. on Logic in Computer Science (LICS), pp. 335–344. IEEE, Piscataway (2004)
42. Dams, D., Namjoshi, K.S.: Automata as abstractions. In: Intl. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI). LNCS, vol. 3385, pp. 216–232. Springer, Heidelberg (2005)
43. De Nicola, R.: Extensional equivalences for transition systems. Acta Inform. **24**(2), 211–237 (1987)
44. De Nicola, R., Vaandrager, F.W.: Three logics for branching bisimulation. J. ACM **42**(2), 458–487 (1995)
45. Gazda, M., Willemse, T.A.C.: Expressiveness and completeness in abstraction. In: Luttik, B., Reniers, M.A. (eds.) Proc. Combined 19th Intl. Workshop on Expressiveness in Concurrency and 9th Workshop on Structured Operational Semantics, EXPRESS/SOS. EPTCS, vol. 89, pp. 49–64 (2012)
46. Godefroid, P., Jagadeesan, R.: Automatic abstraction using generalized model checking. In: Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 2404, pp. 137–150. Springer, Heidelberg (2002)

47. Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. In: Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
48. Grumberg, O.: Abstraction and refinement in model checking. In: Intl. Conf. on Formal Methods for Components and Objects (FMCO). LNCS, vol. 4111. Springer, Heidelberg (2005)
49. Grumberg, O., Lange, M., Leucker, M., Shoham, S.: Don't know in μ-calculus. In: Intl. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI). LNCS. Springer, Heidelberg (2005)
50. Grumberg, O., Lange, M., Leucker, M., Shoham, S.: When not losing is better than winning: abstraction and refinement for the full mu-calculus. Inf. Comput. **205**, 1130–1148 (2007)
51. Grumberg, O., Long, D.E.: Model checking and modular verification. ACM Trans. Program. Lang. Syst. **16**, 843–872 (1994)
52. Grumberg, O., Schuster, A., Yadgar, A.: 3-valued circuit SAT for STE with automatic refinement. In: Intl. Symp. Automated Technology for Verification and Analysis (ATVA). LNCS, vol. 4762. Springer, Heidelberg (2007)
53. Gupta, A., Ganai, M., Yang, Z., Ashar, P.: Iterative abstraction using SAT-based BMC with proof analysis. In: International Conference on Computer Aided Design (ICCAD), p. 416. IEEE, Piscataway (2003)
54. Gupta, A., Strichman, O.: Abstraction refinement for bounded model checking. In: Etessami, K., Rajamani, S. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 3576, pp. 112–124. Springer, Heidelberg (2005)
55. Gurfinkel, A., Chechik, M.: Why waste a perfectly good abstraction? In: Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, pp. 212–226. Springer, Heidelberg (2006)
56. Hennessy, M., Milner, R.: Algebraic laws for nondeterminism and concurrency. J. ACM **32**(1), 137–161 (1985)
57. Henzinger, M.R., Henzinger, T.A., Kopke, P.W.: Computing simulations on finite and infinite graphs. In: FOCS: Foundations of Computer Science, pp. 453–462. IEEE, Piscataway (1995)
58. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: Symp. on Principles of Programming Languages (POPL), pp. 232–244. ACM, New York (2004)
59. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: Jones, N.D., Leroy, X. (eds.) POPL, pp. 232–244. ACM, New York (2004)
60. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), pp. 58–70. ACM, New York (2002)
61. Huth, M., Jagadeesan, R., Schmidt, D.: Modal transition systems: a foundation for three-valued program analysis. In: Sands, D. (ed.) Programming Languages and Systems. LNCS, vol. 2028, pp. 155–169. Springer, Heidelberg (2001)
62. Jain, H., Ivancic, F., Gupta, A., Shlyakhter, I., Wang, C.: Using statically computed invariants inside the predicate abstraction and refinement loop. In: Ball, T., Jones, R.B. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 4144, pp. 137–151. Springer, Heidelberg (2006)
63. Jain, H., Kroening, D., Sharygina, N., Clarke, E.: Word level predicate abstraction and refinement for verifying RTL Verilog. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. **27**, 366–379 (2008)
64. Jhala, R., McMillan, K.: Array abstractions from proofs. In: Damm, W., Hermanns, H. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 4590, pp. 193–206. Springer, Heidelberg (2007)
65. Jhala, R., Podelski, A., Rybalchenko, A.: Predicate abstraction for program verification. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)

66. Kesten, Y., Pnueli, A.: Verification by augmented finitary abstraction. Inf. Comput. **163**(1), 203–243 (2000)
67. Kozen, D.: Results on the propositional μ-calculus. In: 9th Colloquium on Automata, Languages and Programming (ICALP). LNCS, vol. 140, pp. 348–359. Springer, Heidelberg (1982)
68. Kroening, D., Sharygina, N.: Image computation and predicate refinement for RTL Verilog using word level proofs. In: Lauwereins, R., Madsen, J. (eds.) Design, Automation & Test in Europe (DATE), pp. 1325–1330. ACM, New York (2007)
69. Kroening, D., Weissenbacher, G.: Interpolation-based software verification with Wolverine. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV. LNCS, vol. 6806, pp. 573–578. Springer, Heidelberg (2011)
70. Kurshan, R.P.: Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach. Princeton Series in Computer Science. Princeton University Press, Princeton (1994)
71. Lahiri, S.K., Ball, T., Cook, B.: Predicate abstraction via symbolic decision procedures. Log. Methods Comput. Sci. **3**(2) (2007)
72. Larsen, K., Thomsen, B.: A modal process logic. In: Symp. on Logic in Computer Science (LICS), pp. 203–210. IEEE, Piscataway (1988)
73. Larsen, K.G., Xinxin, L.: Equation solving using modal transition systems. In: Symp. on Logic in Computer Science (LICS), pp. 108–117. IEEE, Piscataway (1990)
74. Lee, D., Yannakakis, M.: Online minimization of transition systems (extended abstract). In: Proceedings of the 24th Annual ACM Symposium on Theory of Computing, pp. 264–274. ACM, New York (1992)
75. Li, B., Wang, C., Somenzi, F.: Abstraction refinement in symbolic model checking using satisfiability as the only decision procedure. Int. J. Softw. Tools Technol. Transf. **7**(2), 143–155 (2005)
76. Loiseaux, C., Graf, S., Sifakis, J., Bouajjani, A., Bensalem, S.: Property preserving abstractions for the verification of concurrent systems. Form. Methods Syst. Des. **6**, 11–45 (1995)
77. Mandrykin, M., Mutilin, V., Novikov, E., Khoroshilov, A.V., Shved, P.: Using Linux device drivers for static verification tools benchmarking. Program. Comput. Softw. **38**(5), 245–256 (2012)
78. McMillan, K.L.: Lazy abstraction with interpolants. In: Ball, T., Jones, R.B. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 4144, pp. 123–136. Springer, Heidelberg (2006)
79. McMillan, K.L., Amla, N.: Automatic abstraction without counterexamples. In: Garavel, H., Hatcliff, J. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 2619, pp. 2–17. Springer, Heidelberg (2003)
80. Meller, Y., Grumberg, O., Shoham, S.: A framework for compositional verification of multi-valued systems via abstraction-refinement. In: Liu, Z., Ravn, A.P. (eds.) Intl. Symp. Automated Technology for Verification and Analysis (ATVA). LNCS, vol. 5799, pp. 271–288. Springer, Heidelberg (2009)
81. Milner, R.: An algebraic definition of simulation between programs. In: Intl. Joint Conf. on Artificial Intelligence (IJCAI), pp. 481–489. British Computer Society, London (1971)
82. Namjoshi, K.: Abstraction for branching time properties. In: Hunt, W.A., Somenzi, F. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 2725, pp. 288–300. Springer, Heidelberg (2003)
83. Park, D.: Concurrency and automata on infinite sequences. In: 5th GI-Conference on Theoretical Computer Science. LNCS, vol. 104, pp. 167–183. Springer, Heidelberg (1981)
84. Piterman, N., Pnueli, A.: Temporal logic and fair discrete systems. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
85. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. In: Symp. on Principles of Programming Languages (POPL), pp. 105–118. ACM, New York (1999)

86. Saïdi, H.: Model checking guided abstraction and analysis. In: Palsberg, J. (ed.) Intl. Symp. on Static Analysis (SAS). LNCS, vol. 1824, pp. 377–396. Springer, Heidelberg (2000)
87. Saïdi, H., Shankar, N.: Abstract and model check while you prove. In: Halbwachs, N., Peled, D. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 1633, pp. 443–454. Springer, Heidelberg (1999)
88. Seshia, S.A., Sharygina, N., Tripakis, S.: Modeling for verification. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
89. Sharygina, N., Tonetta, S., Tsitovich, A.: An abstraction refinement approach combining precise and approximated techniques. Int. J. Softw. Tools Technol. Transf. **14**(1), 1–14 (2012)
90. Shoham, S., Grumberg, O.: Monotonic abstraction-refinement for CTL. In: Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 2988, pp. 331–346 (2004)
91. Shoham, S., Grumberg, O.: Multi-valued model checking games. In: Peled, D.A., Tsay, Y.K. (eds.) Intl. Symp. Automated Technology for Verification and Analysis (ATVA). LNCS, vol. 3707, pp. 354–369. Springer, Heidelberg (2005)
92. Shoham, S., Grumberg, O.: Compositional verification and 3-valued abstractions join forces. In: SAS'07. LNCS, vol. 4634. Springer, Heidelberg (2007)
93. Shoham, S., Grumberg, O.: A game-based framework for CTL counterexamples and 3-valued abstraction-refinement. ACM Transactions on Computer Logic (TOCL) **9** (2007)
94. Shoham, S., Grumberg, O.: 3-valued abstraction: more precision at less cost. Inf. Comput. **206**(11), 1313–1333 (2008)
95. Singh, R., Giannakopoulou, D., Pasareanu, C.: Learning component interfaces with may and must abstractions. In: Touili, T., Cook, B., Jackson, P. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 6174, pp. 527–542. Springer, Heidelberg (2010)
96. Stirling, C.: Modal and Temporal Properties of Processes. Texts in Computer Science. Springer, Heidelberg (2001)
97. Tzoref, R., Grumberg, O.: Automatic refinement and vacuity detection for symbolic trajectory evaluation. In: Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 4144, pp. 190–204 (2006)
98. Uribe, T.: Abstraction-based deductive-algorithmic verification of reactive systems. Ph.D. thesis, Computer Science Department, Stanford University (1998). Technical report STAN-CS-TR-99-1618
99. Vizel, Y., Grumberg, O., Shoham, S.: Lazy abstraction and SAT-based reachability in hardware model checking. In: Formal Methods in Computer Aided Design (FMCAD), pp. 173–181. IEEE, Piscataway (2012)
100. Wachter, B., Kroening, D., Ouaknine, J.: Verifying multi-threaded software with impact. In: Formal Methods in Computer Aided Design (FMCAD), pp. 210–217. IEEE, Piscataway (2013)
101. Wikipedia: The free encyclopedia. www.wikipedia.org (2011). Accessed 15 Nov. 2015
102. Witkowski, T., Blanc, N., Kroening, D., Weissenbacher, G.: Model checking concurrent Linux device drivers. In: Intl. Conf. on Automated Software Engineering (ASE), ASE '07, pp. 501–504. ACM, New York (2007)
103. Yahav, E., Reps, T., Sagiv, M.: LTL model checking for systems with unbounded number of dynamically created threads and objects. Tech. Rep. TR-1424, Computer Sciences Department, University of Wisconsin, Madison, WI (2001)

Chapter 14
Interpolation and Model Checking

Kenneth L. McMillan

Abstract In this chapter we consider applications of logical proofs in model checking. Here we are not concerned with using model checking to verify steps in a larger proof but rather with ways in which logical proof methods can aid model checking, particularly in focusing model-checking methods on relevant facts. We introduce a framework for abstraction refinement based on a concept of *deductive generalization*. We then show how various abstraction refinement schemes can be understood in this framework in terms of *local proofs* and *Craig interpolation*. This unifying view exposes the trade-offs made in different systems between the quality and cost of refinements, and also leads to novel model-checking approaches.

14.1 Introduction

In this chapter we consider applications of logical proofs in model checking. Here we are not concerned with using model checking to verify steps in a larger proof (see Chap. 20 of this Handbook [38]) but rather with ways in which logical proof methods can aid model checking, particularly in focusing model-checking methods on relevant facts.

As we saw in Chap. 13 of this Handbook [13], abstraction is crucially important in applying model checking to large and complex systems. Abstraction reduces complexity by focusing on information relevant to a desired property of a system while ignoring irrelevant facts. All known abstraction approaches apply a simple heuristic to determine what is relevant: generalization from particular cases. That is, we consider some simple fragment of the system's behavior (a *case*) and we assume that the steps used to verify this case, suitably generalized, will be relevant to the verification of the full system behavior.

This brings us to the consideration of proofs in model checking. The purpose of the proof in this case is not to act as a certificate of correctness. Rather, we wish to use proofs as *explanations* of correctness that we can use to form generalizations. As a simple example, suppose that we use Bounded Model Checking (see Chap. 10

K.L. McMillan (✉)
Microsoft Research, Redmond, WA, USA
e-mail: kenmcmil@microsoft.com

E.M. Clarke et al. (eds.), *Handbook of Model Checking*,
DOI 10.1007/978-3-319-10575-8_14

of this Handbook [6]) to verify that a property holds within some fixed number of execution steps. Further, suppose we obtain a proof of unsatisfiability of the BMC instance (that is, a proof that the bounded property holds). We may conjecture that the components of the system referred to in this proof are sufficient to verify the property in the unbounded sense. This provides us with a localization abstraction (see Chap. 13 of this Handbook [13]). In effect, we are generalizing an explanation of a case, since we are proposing that the information needed to ensure the property in the bounded case is sufficient for the unbounded case. In fact, a SAT solver can be instrumented to generate a proof of unsatisfiability. Using this proof to generate a localization abstraction is an example of a method called *proof-based abstraction* [34].

There are also substantially less coarse ways of applying proofs as explanations for the purpose of generalization in model checking. Proofs for special cases can be generated by SAT solvers, SMT solvers [5], and other sorts of constraint solvers and theorem provers. These tools can provide proofs for particular cases of system execution, such as bounded executions, control-flow paths, program fragments, and so on. From these proofs, we can derive explanations for correctness of the special cases that can be used to guide abstraction refinement, or other forms of invariant generation, allowing us to verify a property in the general case.

For a proof to act as an explanation, it must meet two key criteria that we will call *utility* and *generality*. The utility criterion requires that explanations be in a form or language that is *usable* in a more general context or proof system. For example, for proofs by invariant, we require explanations in terms of program *state*. By contrast, proofs of bounded executions will likely be expressed in terms of facts about bounded execution *sequences* and therefore unusable. To bridge this gap, we rely on the *Craig interpolation* property of the underlying logic. A *feasible interpolation* result for a given proof system allows us to translate proofs about sequences into proofs about program state, allowing them to be generalized beyond particular execution lengths.

The generality criterion is more heuristic in nature, requiring that explanations for cases actually do generalize, in the sense that they do not rely on specific irrelevant aspects of the case considered, such as the number of iterations of a loop. To achieve this, we rely essentially on Occam's razor. According to this principle, simpler explanations (those employing fewer concepts) are more likely to generalize. The simplicity of the explanation that can be obtained will depend on the proof system (as will the computational difficulty of finding a simple proof). In general, a richer proof system can provide simpler explanations, but may involve greater difficulty in searching the space of proofs.

In this chapter, we will elaborate these two key concepts, and observe that they can explain many of the techniques of abstraction refinement and invariant generation in the literature. We will view these methods as instances of a model of abstraction refinement in which both the abstractor and the refiner are proof systems with different capabilities. We will explain the notion of feasible interpolation and the closely related notion of *local proof*. In this framework, we will observe that most of the extant abstraction refinement methods are actually different local

proof systems that make the key trade-off between simplicity of explanation and computational difficulty in different ways.

14.2 Preliminaries

In this chapter we will use standard unsorted first-order logic, with which we assume the reader is familiar. We assume a signature Σ of function and predicate symbols of defined arities. A symbol with arity zero is called a constant. Terms and formulas are built in the usual way using the symbols of Σ, individual variables, and the standard connectives $\wedge$, $\vee$, $\neg$, $\Rightarrow$, $\forall$, and $\exists$. We write $\top$ for true and $\bot$ for false. We write $\phi[t/a]$ for the result of substituting term t for every occurrence of symbol a in formula ϕ.

A *structure* consists of a universe U and an assignment of functions and predicates over U (of appropriate arity) to each symbol in Σ. A *sentence* is a formula without free variables. We say a structure M is a *model* of a sentence ϕ, written $M \models \phi$, when ϕ is true in M according to the customary semantics of first-order logic.

A *theory* is a set of sentences. We say a sentence ϕ is *valid* in theory T, written $\models_T \phi$, when ϕ is true in all models of T. In this chapter, we will assume a fixed background theory T. Thus, when we say a formula is *valid* or *satisfiable*, we mean respectively that is true in all models of T or some model of T. We say that ϕ entails ψ when $\phi \Rightarrow \psi$ is valid. We will assume that Σ contains a binary predicate symbol $=$ that is interpreted as equality in all models of T.

We will identify a subset Σ_I of the signature Σ that is considered to be *interpreted* by the background theory. This set includes the predicate $=$ and possibly other symbols, such as arithmetic operators. The *vocabulary* of a formula ϕ is the subset of $\Sigma \setminus \Sigma_I$ occurring in it, and is denoted $V(\phi)$. For example, if Σ_I is $\{=, +\}$, then the vocabulary of formula $x = f(y) + z$ is $\{x, f, y, z\}$.

An *inference* consists of a finite sequence of sentences called the *premises* and a sentence called the *conclusion*. An inference is usually written like this:

$$\frac{P_1 \quad \ldots \quad P_k}{C}$$

where $P_1, \ldots, P_k$ are the premises and C is the conclusion. The inference is *sound* when $P_1 \wedge \cdots \wedge P_k$ entails C.

A *derivation* is a sequence of inferences. The conclusion of a derivation is the conclusion of the last inference in the sequence. A formula is a *hypothesis* of the derivation when it occurs as the premise of some inference in the sequence, but is not the conclusion of any earlier inference. A derivation is *closed* if it has no hypotheses. It is easy to see by induction on the derivation length that if the inferences are sound, then the hypotheses entail the conclusion (hence if the derivation is closed, the conclusion is valid).

A proof system $\mathscr{S}$ is a set of rules or patterns whose instances are inferences. A rule with no premises is called an *axiom*. The system is sound when every instance of a rule is a sound inference. An $\mathscr{S}$-derivation is a derivation in which every

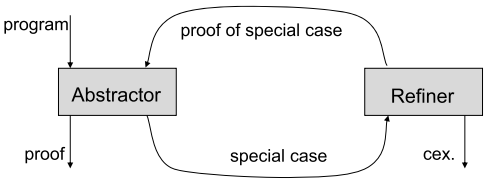


Fig. 1 Abstractor and refiner as proof systems

inference is an instance of a rule in $\mathscr{S}$. A proof system $\mathscr{S}$ is *complete* if every valid formula ϕ is the conclusion of some closed $\mathscr{S}$-derivation.

14.3 Model of Abstraction Refinement

In this section we will introduce an overall perspective on abstraction and refinement that will allow us to step back from the details of particular methods to see the commonalities. This framework will allow us make explicit the trade-offs involved in different approaches. For the sake of simplicity, we will focus only on the verification of safety properties. In all the methods considered, this amounts to inductive invariant generation. Moreover, we will use sequential programs as example systems, primarily because it is easy to give simple expository examples in this form. However, the concepts illustrated are applicable in other domains, such as hardware, protocols, concurrent programs, and so on.

As stated above, we will view abstraction refinement as an interaction of two provers: the abstractor and the refiner. This interaction is diagrammed in Fig. 1. The abstractor is a general prover that is able to prove properties of a class of systems we wish to verify (for example safety properties of sequential programs). However, the abstractor is incomplete, in the sense that it may fail to produce a proof in cases when the property is true. This incompleteness is intentional, its purpose being to restrict the space of proofs to be searched.

The refiner on the other hand is a specialized prover that can only handle certain special cases. For example, the refiner may be restricted only to bounded executions or straight-line programs. Unlike the abstractor, however, the refiner is complete. If the property is not true in the given special case, the refiner can produce a concrete execution in which the property is false. When the abstractor fails to prove a property, it must produce an explanation of its failure in the form of some special case or fragment of the program that it is unable to prove. For example, the abstractor might produce, as an explanation of failure, a particular control-flow path that it cannot verify. It is the refiner's job to produce either a counterexample or a proof for this special case. Abstraction refinement then consists of augmenting the abstractor's proof system so that it can replicate the refiner's proof. In this case, we can say that "refinement progress" has been made, since the abstractor no longer fails on the given case.

Of course, refinement progress is not sufficient to guarantee convergence. We hope that eventually the abstractor will have enough information to prove the pro-

Fig. 2 Hoare semantics of simple program statements

$$\{\psi \Rightarrow \phi\} \quad [\psi] \quad \{\phi\}$$

$$\{\phi[e/x]\} \quad x \coloneqq e \quad \{\phi\}$$

$$\{\forall x\ \phi\} \quad \texttt{havoc}\ x \quad \{\phi\}$$

gram, but except in the finite-state case, it is always possible to diverge by generating an infinite sequence of cases and refinements.

14.3.1 A Simple Programming Language and Proof System

To make these ideas more concrete, we will consider a very simple programming language and its proof system. The language has three kinds of *simple statements*: a *guard*, $[\psi]$, that terminates exactly when formula ψ is true, an *assignment*, $x := e$, that sets variable x to the value of expression e, and a non-deterministic assignment `havoc` x that sets x to a non-deterministic value.

We can define the semantics of our simple programming language in terms of Hoare axioms. These are triples of the form $\{P\}\sigma\{Q\}$, meaning that if P is true of the program state before executing σ, and if σ terminates, then Q is true after executing σ. The Hoare axioms that define the semantics of these statements are shown in Fig. 2.

A *compound statement* is a sequence of simple statements $\sigma_1; \ldots; \sigma_k$. The semantics of the sequence is defined by the usual Hoare logic rule for sequential execution:

$$\frac{\{P\}\sigma_1\{Q\} \quad \{Q\}\sigma_2\{R\}}{\{P\}\sigma_1; \sigma_2\{R\}} .$$

We also admit the *rule of consequence*:

$$\frac{P' \Rightarrow P \quad \{P\}\sigma\{Q\} \quad Q \Rightarrow Q'}{\{P'\}\sigma\{Q'\}} .$$

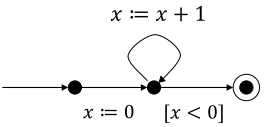
We will model a *program* as a non-deterministic finite automaton (NFA) whose alphabet is the set of compound program statements. Thus, it is a finite-state graph with a defined initial state and a set of final states, whose edges are labeled with sequences of simple statements. The final states represent safety failures. As an example, Fig. 3 shows a simple program with a while loop and its representation as an NFA. The program contains an assertion, which is modeled by a transition to an error state guarded with the negation of the assertion. A string in the language of the NFA is a sequence of statements leading to a safety failure. To prove safety, we must show that no such sequence can actually terminate (which means that always one of the guards must be false). This is what we will ask our abstractor to prove.

The abstractor will construct proofs by inductive invariant. Suppose we fix a logical language L (a subset of first-order logic) whose vocabulary consists of the

Fig. 3 A simple program and its control-flow automaton

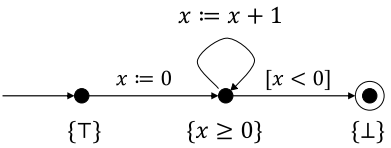
```
x = 0;
while(*)
    x++;
assert x >= 0;
```

(a)



(b)

Fig. 4 Safety invariant of simple program



program variables. A *safety invariant* for program C is a map from the vertices of C to formulas in L such that

1. The initial vertex is labeled $\top$,
2. The final vertices are labeled $\bot$, and
3. Each arc in the graph is a provable Hoare triple.

The last condition means that the pre-condition of each arc guarantees the post-condition after executing the given statement. Existence of a safety invariant guarantees that the final vertices are not reachable from the initial vertex.

As an example, Fig. 4 shows a safety invariant of the program of Fig. 3.

14.3.2 The Abstractor

Whether or not our abstractor can produce this proof depends on whether the language L contains the predicate $x \geq 0$. If not, we would like the abstractor to produce a case that it cannot prove. Under some assumptions, we can show that a program path always suffices for this purpose. That is, if there is no safety invariant for a program L then there is an accepting run of the program automaton that also has no safety invariant in L. We will refer to this property as *path-reductiveness*. It says that absence of a proof can always be explained with reference to a single program path. We can show that this property holds if L is finite and closed under conjunction and disjunction.[1]

[1]The proof of this is simple, though not directly relevant to our discussion. The key is that the language L is closed under arbitrary conjunctions and disjunctions. This means that every program has a strongest inductive invariant in L, which is the conjunction of all the inductive invariants. In particular, every finite path has a strongest inductive invariant (we can obtain it by computing the strongest post-condition in L along the path). The disjunction over all paths of the strongest inductive invariants is an inductive invariant of the program, necessarily the strongest. If a final state is not labeled with a formula equivalent to $\bot$ in the strongest inductive invariant, there is a finite path in which this state is also not labeled $\bot$.

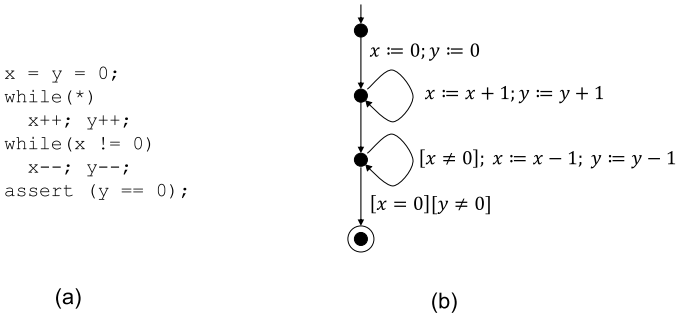


Fig. 5 Increment/decrement program and control-flow automaton

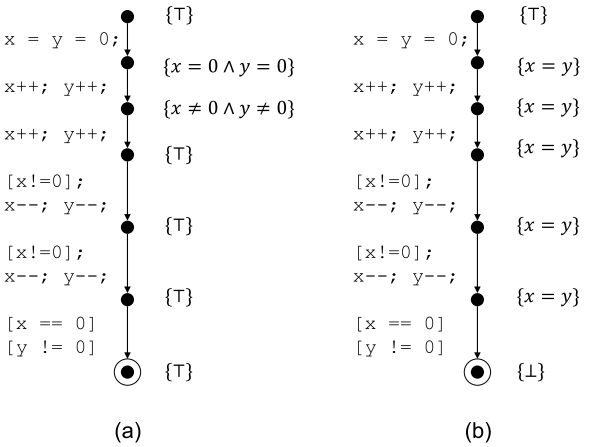


Fig. 6 Failure path in increment/decrement program

As an example, consider the program and corresponding automaton shown in Fig. 5. In this program, two variables x and y are initialized to zero. In the first loop, while a non-deterministic condition holds, both are incremented. In the second loop, while x is non-zero, both are decremented. When this loop exits, we assert the safety condition that y is zero. Now suppose we are using predicate abstraction to prove this property (see Chap. 15 of this Handbook [22]). We begin with a set of atomic predicates P containing $x = 0$ and $y = 0$. From our perspective, predicate abstraction with predicates P means that our logical language L is the set of Boolean combinations over P. That is, the abstractor can prove the property exactly when there is a safety invariant expressible as a Boolean combination of predicates in P. In this case, however, no such proof exists, so the abstractor must fail. Since our language L is finite and closed under conjunction and disjunction, our abstractor is path-reductive. This means that we can exhibit a single program path that has no proof in language L. One such path is shown in Fig. 6(a). In this path, both loops have been executed twice. The path has been labeled with its strongest inductive invariant expressible in L. Notice that after one iteration of the loop, we can infer

that both variables are non-zero, but after two iterations, we can infer nothing in L (except $\top$). As a result, we fail to prove $\bot$ at the error state, hence this path provides an explanation of the failure of the abstractor.

14.3.3 The Refiner

This case can be given to a refiner. The refiner is a specialized prover that can handle only single-path programs. If the case is safe, the refiner can produce a proof as an explanation of the case to the abstractor. Such a proof is shown in Fig. 6(b). It uses an atomic predicate $x = y$ that is not in the abstractor's language L. Given this explanation, the abstractor's proof system can be augmented by adding the predicate $x = y$ to P. The abstractor using L can now replicate the refiner's proof for this case (in fact it can generate a slightly stronger proof, since it can infer that $x \neq 0$ and $y \neq 0$ after the first loop iteration). Hence, after this refinement, the abstractor will never produce this particular path as a failure case and we can say we have made refinement progress.

In this case, augmenting the abstractor's proof system was simple. This might not always be the case, however. We have assumed a class of possible abstractor proof systems, parameterized over a class of finite logical languages. Moreover, we have assumed that the abstractor and refiner have equal deductive power over the same set of formulas. Thus in the case of predicate abstraction it is easy to augment the language of the abstractor to allow it to replicate the refiner's proof, by simply adding the atomic predicates that occur in the refiner's proof (assuming it is quantifier-free, or that we allow quantified predicates in P).

Consider as an alternative, however, the Cartesian abstraction [2]. This abstractor can only deduce post-conditions of statements that are literals over a fixed set of predicates P. As a result, it can only replicate path proofs that consist of cubes over P (conjunctions of literals).[2] If the refiner's proof contains a disjunction, we would have to treat the disjunction as an atomic predicate.

In general, it might be a non-trivial problem to find a small (let alone minimal) augmentation to the abstractor's proof system that replicates the refiner's proof.

14.4 Refinement, Local Proofs, and Interpolants

Now we will consider the problem of creating refiners in the above schema. Fundamentally, in this approach we are producing generalizations from explanations of cases. That is, the refiner produces the explanations, in the form of Hoare proofs

[2]Using disjunctive completion, disjunctions may occur as the result of joining multiple paths, but they do not occur in single path proofs. Disjunctive completion is needed for path-reductiveness, and in fact is used in tools such as SLAM [1, 3] that refine Cartesian abstractions.

for cases, and the abstractor generalizes these explanations by producing an inductive invariant from aspects of the refiner's proof. One way to think of this is that the refiner has produced a vocabulary of deductive steps from which the abstractor attempts to assemble a proof in a more general setting.

From this point of view, we can see immediately that there are two important criteria for the refiner's explanation. First, it must consist of steps that are meaningful in the abstractor's framework (the *utility* criterion). Second, the refiner's proof must generalize, in the sense that it may not rely on deductions that can only be made in the particular case at hand, but must be useful in the more general setting (the *generality criterion*).

14.4.1 Craig Interpolation

To meet the first criterion in our example we have required that the refiner produce explanations in the form of Hoare logic proofs. However, standard verification methods (i.e., bounded model checking or verification condition checking) will not produce proofs in this form. In these methods, the proof is achieved by translating the proposed error path into a set of logical constraints (formulas) and using some decision procedure (e.g., SAT or SMT) to show infeasibility of these constraints. These decision procedures can potentially produce proofs, but the proofs will not generally be in the desired form. To obtain proofs in a form that is useful to the abstractor, we rely on the *Craig interpolation* property of the underlying logic.

Craig's interpolation lemma [12] says that, given two formulas A and B in first-order logic, if $A \Rightarrow B$ is valid, then there exists a formula I such that $A \Rightarrow I$ and $I \Rightarrow B$, where I is expressed using the common vocabulary of A and B. That is, we must have $V(I) \subseteq V(A) \cap V(B)$. This means that any symbol occurring in I that is not interpreted must occur in *both* A and B. The formula I is referred to as an *interpolant* for the implication $A \Rightarrow B$. The interpolation lemma applies to plain first-order logic and to first-order logic with interpreted equality. Other theories may or may not have the interpolation property.

The interpolant formula I in effect gives us a modular proof of $A \Rightarrow B$. That is, we can show $A \Rightarrow I$ reasoning only in the vocabulary of A and $I \Rightarrow B$ reasoning only in the vocabulary of B. Thus, finding an interpolant is essentially factoring the proof of $A \Rightarrow B$ into a modular form. We will see that modularity has important consequences for abstraction refinement.

Of course, the existence of an interpolant is of little use if we can't find one or if the formula is too complex. It turns out, however, that a number of useful theories and proof systems have the property of *feasible interpolation*. This means in effect that if we can find a non-modular proof of $A \Rightarrow B$, we can obtain an interpolant (hence a modular proof) in polynomial time. To be more precise, a proof system $\mathscr{S}$ has the *feasible interpolation* property if, given a closed $\mathscr{S}$-derivation of $A \Rightarrow B$, we can derive an interpolant for $A \Rightarrow B$ in polynomial time. Different proof systems for the same theory may or may not allow feasible interpolation.

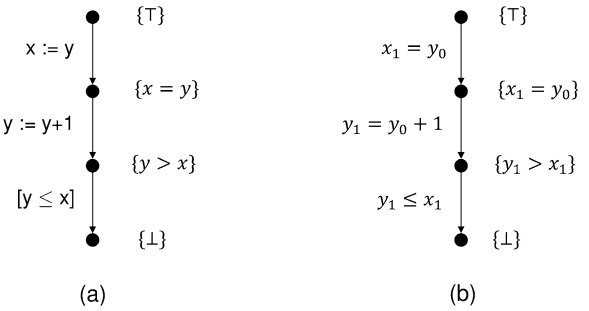


Fig. 7 Short program path (**a**) and its SSA form (**b**)

The proof systems we will be dealing with are *refutation* systems, meaning they are designed to derive $\bot$ from a conjunction of hypotheses. For such proof systems, it is more convenient to speak of an interpolant for an inconsistent conjunction. That is, an interpolant I for an inconsistent conjunction of formulas $A \wedge B$ is a formula over the common vocabulary of A and B such that $A \Rightarrow I$ and $I \Rightarrow \neg B$. The convention of negating B when dealing with refutation systems was introduced by McMillan [28] and is common in the literature on interpolation in model checking. A refutation system has the feasible interpolation property if an interpolant for $A \wedge B$ can be derived from a refutation from premises A and B in polynomial time.

Feasible interpolation provides of way of deriving Hoare logic proofs about program paths from the proofs generated by decision procedures. One way to view this is that Craig interpolants are precisely Hoare logic proofs for program paths in *static single assignment* (SSA) form. To obtain this result, we need a simple generalization of the notion of Craig interpolant to formula sequences.

As an example, consider the short program path in Fig. 7. This path is not feasible, since, upon reaching the guard $[y \leq x]$, we have $y = x + 1$. We can prove this by converting the path into a sequence of logical constraints in SSA form. To do this, we rewrite each assignment to a logical equality by giving the assigned variable a fresh subscript. For example the assignment $y := y + 1$ could become the logical constraint $y_1 = y_0 + 1$. Note that in this form, each variable is "assigned" only once, thus we can think of the assignment as logical equality. The resulting set of constraints is feasible exactly when the original program path can terminate (i.e., when there is an assignment of initial values to the variables that makes all the guards true). We can now ask a decision procedure whether the set of SSA constraints is satisfiable. If not, a proof-producing decision procedure will produce a refutation in a suitable proof system. This is not a Hoare logic proof, since, for example, it may contain predicates that mix variables that represent different program states, it may reason both backward and forward in time, and so forth. The root of this problem is that the decision procedure is reasoning about the entire path rather than individual program states.

Given a feasible interpolation result, however, we can always translate this non-modular proof into a modular proof. An interpolant for our example is shown in Fig. 7(b). This is a sequence of predicates beginning with $\top$ and ending with $\bot$, such that each predicate implies the next (given the corresponding constraint) and

each is written in the common vocabulary between the preceding and following constraints. This is known as a *sequence interpolant* and was introduced in [18]. The sequence interpolant has the property that each predicate is written using the variable instances that are "in scope" at a particular point in the path, and thus represent the program state at that point. The interpolants thus form a Hoare refutation of the SSA sequence. The reader can verify that simply dropping the subscripts from the variables yields a Hoare logic refutation of the original path.

14.4.2 Feasible Interpolation in Model Checking

The fact that interpolants are Hoare proofs for SSA programs (or equivalently for BMC unfoldings) allows us to build refiners from various decision procedures, including SAT and SMT solvers. This idea was originally used to allow finite-state model checking using only a proof-producing SAT solver [28]. This method constructs a BMC unfolding of a circuit to depth k. A SAT solver is used to refute the unfolding formula (i.e., the SSA form). From the resulting proof using the resolution rule, a feasible interpolation procedure is used to construct an interpolant (i.e., a Hoare proof) in linear time. Various methods can then be used to attempt to construct an inductive invariant from the interpolant. In the original method, the one-step interpolant is iterated until it becomes inductive. Intuitively, this iteration gives the SAT solver many chances to generalize (to eliminate irrelevant information applying only to short paths). It is shown that for large enough k, the solver may not overgeneralize, and therefore must eventually converge to an inductive invariant, yielding a complete model-checking procedure using only SAT. Other kinds of abstractors are also possible. For example, it is possible to construct inductive invariants as conjunctions or disjunctions of the interpolant formulas [30, 32].

Proofs generated from SMT solvers can also be translated into interpolants for various theories. For example, McMillan gave an interpolation calculus for linear rational arithmetic with uninterpreted functions [29]. This was used as a refiner for predicate abstraction in the BLAST software model checker [18]. The system has the property that it produces quantifier-free interpolants from quantifier-free formulas. We will observe shortly the advantage of this property.

This scheme of interpolating proofs from SMT solvers has been extended in various ways. Yorsh and Musuvathi describe a general framework for interpolation with combined theories that are *equality interpolating* [40]. Interpolation calculi have been extended to richer theories, including integer arithmetic [7, 10, 16, 20, 25], the theory of arrays and sets [9, 21, 23] and bit vectors [15, 27]. Some schemes also handle logics with quantifiers [8, 31, 33].

14.4.3 Interpolants and Local Proofs

An interpolant can be viewed as an example of a *local proof* [31]. In a local proof, we define a set of local vocabularies and we restrict every inference to use just one

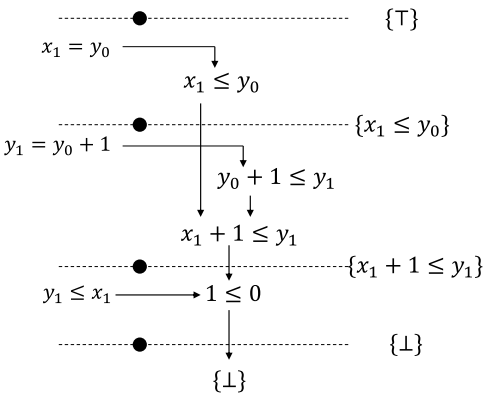


Fig. 8 Flow of a forward local proof, providing an interpolant

local vocabulary. We can think of feasible interpolation as rewriting a proof into a local form. That is, an interpolant I for $A \wedge B$ provides us with a cut in the proof, allowing us to reason locally (i.e., we can prove $A \Rightarrow I$ only in the vocabulary of A and $I \Rightarrow \neg B$ only in the vocabulary of B). Conversely, local proofs provide us a means of producing interpolants. By applying feasible interpolation results for resolution proofs, we can show that any local refutation for $A \wedge B$ can be translated into an interpolant for A and B in linear time. This interpolant can be formed as a Boolean combination of formulas occurring in the local proof [21].

In this section, we will define the notion of local proof, and observe that many of the refiners described in the literature can be viewed as specific local proof systems, with the refinements being the resulting interpolants.

As an example, consider the sequence of SSA constraints in Fig. 7(b). We will call each constraint in this sequence a "frame". In a local refutation, we might use the following sequence of inferences. From $x_1 = y_0$, we infer $x_1 \le y_0$. This inference is local to the first frame. From this, and $y_1 = y_0 + 1$, we derive $x_1 + 1 \le y_1$ by summing inequalities. This inference is local to the second frame. Summing this with $y_1 \le x_1$, we obtain $1 \le 0$, a contradiction. Figure 8 shows the flow of inferred predicates in this proof. That is, we have drawn an arrow from each premise of an inference to its conclusion. In our proof, these arrows cross the frame boundaries only in the forward direction. Moreover, the predicates that are passed across the boundary use only the symbols that are "in scope" at the boundary. The result is that these predicates (shown on the right) form an interpolant. Note there are various possible proofs that result in different possible interpolants (for example, using equalities rather than inequalities).

Now let's define more precisely what we mean by a local proof. Suppose we are given a sequence of frames $\sigma = \phi_1 \dots \phi_k$. We will say that a symbol is *in scope* in frame i when it occurs in the vocabularies of some ϕ_l and some ϕ_h such that $l \le i \le h$. In our example, the variables in scope in the successive frames are $\{x_1, y_0\}$, $\{x_1, y_0, y_1\}$ and $\{x_1, y_1\}$. Notice that when we assign a variable in a given frame, both the new and the old instance of that variable may be in scope. Notice also

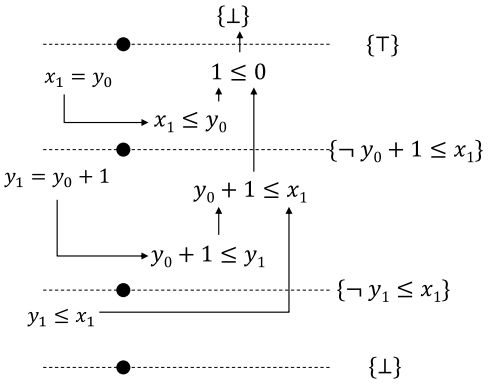


Fig. 9 Flow of a reverse local proof, negated, providing an interpolant

that the intersection of the scopes of two consecutive frames is precisely the set of variables carrying the program state between those frames.

We will say that the vocabulary of an inference is the union of the vocabularies of its premises and its conclusion. An inference is *local* for σ if its vocabulary is in scope at some frame of σ. A derivation is local if all of its inferences are local. The reader can confirm that in Fig. 8, each inference is local to the frame in which we have placed its conclusion.

We can also distinguish *forward local* and *reverse local* proofs. In a forward local proof, we can assign each derivation step to a frame, such that the flow of information is only forward. That is, an inference's premises may not be conclusions or hypotheses from later frames. Conversely, in a reverse local proof, the flow of information is only backward (so no inference uses a conclusion or hypothesis from an earlier frame).

Above we saw an example of a forward local proof. The facts flowing forward in this proof formed an interpolant. Consider on the other hand a reverse local proof. First, from $y_1 \leq x_1$ and $y_1 = y_0 + 1$, we derive $y_0 + 1 \leq x_1$. This we can assign to the second frame (but not the third, since it uses y_0). From this and $x_1 = y_0$, we derive the contradiction $1 \leq 0$. This we can assign to the first frame (but not the second, since it uses a hypothesis from the first). Figure 9 shows the information flow in this proof. Reverse proofs are the De Morgan dual of forward proofs. That is, an interpolant can be obtained as the *negation* of the facts flowing backward.

Now we will define what we mean by flow of premises in a proof, and consider the general case of local proofs. We assume that each inference s in a derivation P has been assigned to a frame $f(s)$ in which s is local. Consider the boundary between consecutive frames i and $i + 1$. We can divide the derivation P into two sub-sequences. The first, $P_{\leq i}$, consists of the inferences of P assigned to frames before the boundary and the second, $P_{>i}$, consists of those assigned to frames after the boundary. For convenience, we will add a special terminal step to P, which has the conclusion $\bot$ and is assigned to the imaginary frame $k + 1$.

The *forward flow* at the boundary is the set of conclusions of inferences in $P_{\leq i}$ or frames $\leq i$ that are used as hypotheses of $P_{>i}$. Conversely, the *reverse flow* is the set of conclusions from $P_{>i}$ or frames $> i$ that are used as hypotheses of $P_{\leq i}$. Now

consider any formula ψ in the forward flow. In proof $P_{\leq i}$, this conclusion depends on some subset $\text{dep}(\psi)$ of the reverse flow. Thus, the frames before the boundary can be used to prove $\text{dep}(\psi) \Rightarrow \psi$. Note that this fact is in scope in both frames i and $i+1$, so it uses the common vocabulary. Moreover, from these implications in $P_{>i}$, we can construct a proof of $\bot$. Thus, if we collect the conjunction of $\text{dep}(\psi) \Rightarrow \psi$ for each ψ in the forward flow, we have an interpolant.

Note that in the forward local case, the sets $\text{dep}(\psi)$ are empty. Thus the interpolant is just the conjunction of the formulas in the forward flow. In the reverse local case, the only formula in the forward flow is $\bot$, so the interpolants are effectively the negation of the reverse flow. In the general case, however, the interpolant might be a conjunction of Horn clauses over the formulas in the proof.

This flow-based approach to generating interpolants from local proofs was introduced by Kovács and Voronkov [24]. The interpolants generated are in the worst case quadratic in the proof size, since each implication is linear in the reverse flow size and the number of implications is linear in the forward flow size. By applying feasible interpolation methods for resolution proofs, Jhala and McMillan showed that linear-size interpolants can be obtained [21].

This result means that we can think of Craig interpolants as a kind of normal form for local proofs. That is, an interpolant is the forward flow of a local proof, and every local proof can be translated into an interpolant. Moreover, if a proof system has the feasible interpolation property, we can translate *any* proof in the system into a local form.

14.5 Refiners as Local Proof Systems

Now we will consider several refiners from the literature and show that they can be viewed as local proof systems.

14.5.1 Strongest Post-condition

The SLAM model checker [1, 3] was an early implementation of software model checking based on predicate abstraction and has been highly influential on subsequent systems. SLAM's refiner, Newton [4], is based on the computation of *strongest post-conditions*. Given a formula ϕ and a program statement σ, the strongest post-condition (SP) of ϕ with respect to σ is the strongest ψ such that $\{\phi\}\ \sigma\ \{\psi\}$ is a valid Hoare triple. This can also be viewed as the set of states reachable from ϕ by executing σ, or the forward image of ϕ with respect to σ, if we view σ as a transition relation. As an example, if we start with $\{\top\}$ and compute strongest post-conditions for the path of Fig. 7(a), we get the sequence $\{\top\}, \{x = y\}, \{y = x + 1\}, \{\bot\}$. Since this sequence ends with $\{\bot\}$, it is a Hoare refutation of the path.

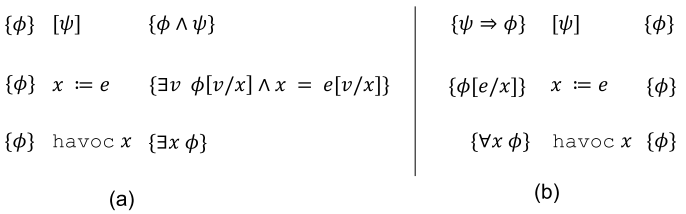


Fig. 10 Rules for SP (**a**) and WP (**b**)

Fig. 11 Local proof system for SP and WP

$$\frac{\phi \quad x = e}{\phi[e/x]}\ x \text{ max in } \phi, e \qquad \frac{\Gamma}{\bot}\ \Gamma \text{ unsat.}$$

Strongest post-conditions can be computed syntactically. For our simple language, the axioms in Fig. 10(a) suffice. This computation is also known as *symbolic execution*. For SSA paths, we can also think of the syntactic SP computation as a simple forward local proof system based on ordered rewriting. That is, let us define a total *elimination order* $\succ$ that puts the SSA variables in the order in which they go out of scope. In our example, we would have $y_0 \succ x_1 \succ y_1$. Now consider the local proof system of Fig. 11. There is one rule that allows replacement of equals with equals in a formula. However, a side condition states that we replace only the earliest SSA variable occurring in either premise. We may also deduce a contradiction, provided that all the premises are local. Finally, we require that in the SSA form, each occurrence of `havoc` v be replaced by the equation $v = \Theta_i$ where Θ_i is a fresh logical constant symbol. We can think of the Θ_i symbols as representing symbolic "inputs" to the program, and the proof system as a rewrite system that computes the state of the program at each step as a function of these inputs. That is, if we view the substitutions as rewrites (*replacing* ϕ with $\phi[e/v]$) then we have a confluent and terminating system that reduces the SSA program P to a normal form $N(P)$ consisting of a set of equations $v = e$, where v is an SSA variable and e is over just the inputs, and guards expressed only in terms of inputs. This normal form represents a set of program states: the valuations of the variables that make $N(P)$ true for *some* valuation of the inputs, or $\exists\Theta.\ N(P)$.

Moreover, the requirement that the rewrites respect the elimination order guarantees that the proofs are forward local. That is, we can assign each rewrite the later of the frames of its two premises, ensuring forward information flow in the proof. After rewriting terminates, the forward flow of the proof at each boundary is precisely the strongest post-condition at that boundary (i.e., projected onto the variables in scope at the boundary).

As an example, Fig. 12 shows the derivation obtained by terminal rewriting for our short example, and the corresponding forward flow. After existentially quantifying the input symbols, we obtain the interpolant sequence shown on the right in the figure. Eliminating the quantifiers, we obtain the equivalent formulas $\top$, $x_1 = y_0$, $y_1 = x_1 + 1$, $\bot$, that is, precisely the sequence of strongest post-conditions we ex-

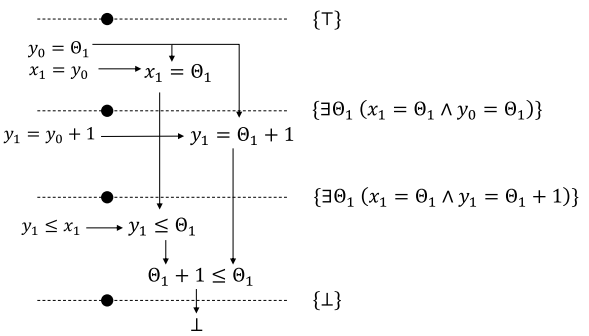


Fig. 12 Forward flow for proof using SP system

pect. This gives us a Hoare proof of a particular path, which can act as a refinement in predicate abstraction.

Using SP as a refiner, though it has the advantage of simplicity, leaves much to be desired from the point of view of utility and generality. Though SP produces Hoare proofs, it introduces existential quantifiers over the input symbols. This is a significant utility issue when using predicate abstraction, for two reasons. First, predicate abstraction can synthesize only Boolean combinations of the given predicates P, and not quantifiers. This means that a quantified formula must be treated as atomic for predicate abstraction. Second, quantifiers in the predicates make the decision problems involved in predicate abstraction undecidable, and in practice much more difficult for SMT solvers. In our simple example, the quantifiers were easily eliminated. However, in general the underlying theory may not admit quantifier elimination, or quantifier elimination may produce intractably large formulas (for linear rational arithmetic, the result is doubly exponential). The original refiner in SLAM tried to finesse this issue by, in effect, introducing auxiliary program variables that witness the existential quantifiers (a subject that is beyond the scope of this chapter). Ultimately, however, this is a futile effort. One easily constructs loops for which the number of such variables diverges to infinity.

A related difficulty is one of generality. That is, the strongest post-condition, by its very definition, fails to generalize away from irrelevant details of a path. For this reason, real refiners (including SLAM) do not actually compute the strongest post-condition. Rather they weaken SP in an attempt to eliminate irrelevant predicates. This weakening is easily understood by viewing SP as a local proof system.

That is, when constructing a proof, it is natural to eliminate steps that are possible, but on which the ultimate conclusion does not depend. Moreover, we require only one derivation of any given fact, and may eliminate redundant derivations. If one performs this natural reduction on proofs generated by the SP system of Fig. 11, the result is to eliminate irrelevant facts from the forward flow, and hence from the interpolant. In fact, SLAM does precisely this to improve the generality of refinements.

Another natural way to improve the generality of a proof is to weaken the hypotheses. In the case of SSA programs, one can, for example, replace an assignment $v = e$, where e is a complex but potentially irrelevant expression, by $v = \Theta_i$. Since

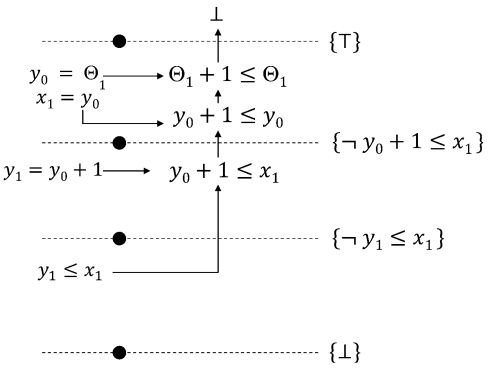


Fig. 13 Reverse flow for proof using WP system, negated

Θ_i is implicitly existentially quantified, this is a weaker hypothesis. In fact, this kind of weakening can greatly improve the generality of the interpolants, in some cases making it possible to abstract away from the number of iterations of a loop. SLAM and other model checkers perform this step.

14.5.2 Weakest Pre-condition

The time-reversal dual of strongest post-condition is weakest pre-condition (WP). Given a formula ϕ and a program statement σ, the weakest (liberal) pre-condition of ϕ with respect to σ is the weakest ψ such that $\{\psi\}\ \sigma\ \{\phi\}$ is a valid Hoare triple. This can also be viewed as the set of states that cannot reach $\neg\phi$ by executing σ. As an example, if we start with $\{\bot\}$ at the path end and compute weakest pre-conditions backward for the path of Fig. 7(a), we get the sequence $\{True\}, \{x < y + 1\}$, $\{x < y\}, \{\bot\}$, another possible Hoare refutation of the path, using weaker facts.

Weakest pre-conditions can be computed syntactically. For our simple language, the axioms in Fig. 10(b) suffice. Note that when computing weakest precondition, we may introduce a universal quantifier, but only in case of a `havoc` statement. Given the difficulties involved in quantifiers, this may be an advantage for WP, relative to SP.

The WP computation may also be viewed as a local proof system. In fact, it is exactly the proof system of Fig. 11, except that the *elimination* order is replaced by an *introduction* order. For our example, we have $y_1 \succ x_1 \succ y_0$, the reverse of the order in which the variables are introduced (or their order of elimination in a reverse execution). Again, we must encode `havoc` v by the equation $v = \Theta_i$ where Θ_i is a fresh logical constant symbol.

The time reversal of the elimination order results in a *reverse* local proof (where each step is assigned to the frame of its *latest* premise). After rewriting terminates, the *reverse* flow of the proof at each boundary is precisely the *negation* of the weakest pre-condition of $\bot$.

As an example, Fig. 13 shows the derivation obtained by terminal reverse rewriting for our example, and the corresponding reverse flow. Taking the negation of the

reverse flow, we obtain the interpolant sequence shown on the right in the figure. Modulo subscripts, this is equivalent to the WP sequence we obtained above. In this case, we were lucky. Because the only input occurred in the first frame, there were no quantified input variables in the interpolant. In general, though, the input variables will occur existentially quantified in the reverse flow, and thus after negation will be universally quantified in prenex form. Note that proof reductions and hypothesis weakening can also be applied to WP proofs to improve the generality of the interpolants. Moreover, there is no reason why the SP and WP proof systems cannot be mixed (allowing both forward and reverse rewriting). The result will be a general local proof, which can still be interpolated. Finally, note that for SP, the interpolants are conjunctions of the facts in the flow, while for WP they are disjunctions (because of the negation). This may make SP better suited as a refiner for the Cartesian abstractor (used in SLAM), since the latter can only reproduce conjunctions in path proofs.

14.5.3 Bounded Provers

The SP and WP proof systems are quite weak (allowing only substitution of equals for equals). As a result, the local proofs they produce tend not to generalize even in simple cases. Consider, for example, the simple program of Fig. 5 and the path of this program illustrated in Fig. 6. As we saw, the simple predicate $x = y$ is sufficient to construct an inductive invariant for this program. However, to do this, our refinement proof must abstract away from a particular number of executions of the loop. For the example path, both SP and WP yield the interpolant sequence $\{\top\}, \{y = 0\}, \{y = 1\}, \{y = 2\}, \{y = 1\}, \{y = 0\}, \{\bot\}$ after proof reduction. Clearly, these predicates are only relevant to the case of exactly two loop iterations, and do not generalize to longer paths. Moreover, there is no obvious way to weaken these proofs to promote generality, since we know x must somehow be involved in the inductive invariant. Thus, using SP or WP as a refiner, we will diverge, generating an infinite sequence of refinements, for longer and longer paths.

The failure here can be put down to the weakness of the proof system and a failure to apply Occam's razor. That is, we need a refinement proof that is not only in the right form (meeting the utility criterion) but that is in some sense simple, using as few constructs as possible in order to have a better chance to generalize. In our example, a simpler proof would use only the predicate $x = y$.

A straightforward approach to this is to search for the simplest proof according to some metric. This approach was introduced in [26] in the context of abstraction refinement for hardware verification.

It uses a so-called *bounded prover*, defined by the following two components:

1. A local proof system, defined by an *ad hoc* set of proof rules, and
2. A cost metric for proofs.

For the cost metric, we might count, say, the number of distinct atomic predicates in the proof after dropping subscripts. Note this means that our notion of simplicity is relative to the intended use of the proof.

We can now simply search the space of possible proofs for a proof of lowest cost. In performing this search, we can take a branch-and-bound approach, since each successful proof gives an upper bound on the cost.

As an example of such a system, consider the system of Fig. 11 with two modifications: we remove the ordering constraint on substitutions, and we allow solving an arithmetic equation for a chosen variable. Now consider two possible refutations for the path of Fig. 6 using this system. The first uses $y_0 = 0$ to rewrite $y_1 = y_0 + 1$ in the second frame, yielding $y_1 = 1$. Similarly, in the next frame, we have $y_2 = 2$, then $y_3 = 1$, $y_4 = 0$ and finally $0 \neq 0$ in the last frame. If we expand the example path to N iterations of the loop, this proof has a cost of $N + 1$, since it generates $N + 1$ distinct predicates. On the other hand, consider a proof in which $y_0 = 0$ is used to rewrite $x_0 = 0$, obtaining $x_0 = y_0$. In the second frame, this rewrites $x_1 = x_0 + 1$ to obtain $x_1 = y_0 + 1$. Solving for y_0, we have $y_0 = x_1 - 1$, which then rewrites $y_1 = y_0 + 1$ to yield $x_1 = y_1$, and so on. The number of *distinct* predicates we obtain in this proof, ignoring subscripts, is just four for any N. Thus, for a long enough path, we prefer the second proof. Note that the simpler proof (in terms of predicates used) generalizes to an inductive invariant. However, the SP and WP systems are too restricted to yield this proof.

There are trade-offs to be made in such a system. A richer proof system may yield a simpler proof, but the cost of searching for the proof may be higher. Moreover, the proof system itself constitutes an inductive bias, since it determines what facts have simple proofs. Thus, the choice of proof rules might provide a means of introducing domain knowledge into the system (see, for example, [31]). We saw in the case of WP and SP that a very simple proof system can be complete (given the local contradiction rule). We might need a much richer system, however, to produce generalizations in a given domain.

Compromises can also be made. For example, the method of [26] doesn't find a globally optimal proof. Rather, it finds optimal proofs only for particular truth assignments to the atomic propositions, rather in the style of an SMT solver. This trades off quality of the proof for a considerable reduction in cost of the search.

14.5.4 Split Provers and Language Stratification

An alternative approach to applying Occam's razor uses local proof search in a different way [21]. In the "split prover" approach, an infinite logical language L is stratified into a hierarchy of finite languages $L_1 \subseteq L_2 \cdots$. For example, language L_k might be restricted to terms of nesting depth k, perhaps using only some fixed set of constructors for constants. We can guarantee to find a proof for a given path in the lowest possible L_k (i.e., the simplest language) provided the local prover has a property called *completeness for consequence finding*. This means that, given a set

Γ of formulas in L_k that are in scope at frame i, we must be able to infer all the consequences of Γ expressible in L_k that are in scope in frame $i+1$. The stratification of the language constitutes a very explicit inductive bias, and is arguably less natural than a simple proof cost metric. On the other hand, this approach provides a kind of completeness guarantee. That is, if there is an inductive invariant proving the property in language L, then the refinement sequence is guaranteed to eventually produce one.

14.5.5 Constraint-Based Interpolation

Another approach to searching for simple proofs is to define a class of proofs and pose the question of existence of a proof as a constraint-solving problem. One such approach [37] is based on Farkas' lemma. This result states that if a set of linear inequalities over the rationals is inconsistent, then one can obtain a refutation by simply summing up the inequalities with non-negative coefficients, to obtain the result $0 \leq -1$.

It is easily shown that Farkas proofs are local [36]. That is, given two sets A and B of inequalities, if we order the summation so that the inequalities from A are added first, then the intermediate sum we obtain is only over the common variables (that is, if it contained a variable not occurring in B, we would be unable to cancel it out by adding inequalities from B). It follows that the intermediate sum is an interpolant.

It is well known that Farkas proofs can be obtained by setting up a linear programming (LP) problem in which the Farkas coefficients are the variables. This allows us to search the space of proofs using LP techniques. Since the coefficients in the interpolants are just linear functions of the Farkas proof coefficients, we can attach constraints or objective functions to these coefficients. For example, we could even require that the interpolant coefficients at two given points in the path be the same, if this helps to construct an invariant for a loop. We can also increase the space of proofs by adding any valid inequalities to the set (for example, adding $0 \leq 1$ to each frame would allow us to find the inductive invariant in Fig. 6).

This approach is limited, in the sense that it only handles linear inequalities over the rationals. However, it provides a good example of trading off the cost of searching for an optimal proof against the richness of the proof system.

14.5.6 Feasible Interpolation and Refinement

As we have seen, many methods of refinement can be viewed as search for a proof in a suitable local proof system. We have also observed that there is a trade-off between cost of the search and quality of the proof, which in turn determines the likelihood that a proof will generalize. Proof search in the SP and WP systems is quite straightforward (we simply run rewriting to termination and check the resulting formulas for

satisfiability with a decision procedure). However, the poverty of the proof system can result in proofs that do not generalize away from features specific to a given case, such as number of iterations. A richer deduction system can produce a higher quality proof, but at a cost.

This also puts the approach of feasible interpolation in context. This method converts a non-local proof to a local proof. It is better than the SP and WP systems in terms of utility, as long as we have quantifier-free interpolation, since it does not unnecessarily introduce quantifiers. However, from the generality point of view, it relies on a decision procedure's inbuilt heuristics to generate a simple proof. This can be far from optimal, as a simple proof in the decision procedure's system may be much less so after interpolation. Thus, the feasible interpolation approach makes proof search relatively easy (allowing efficient SAT and SMT solvers to be used) but at the possible expense of generality of the result.

14.5.7 Improving Interpolants

For a given proof, there are various ways in which we can adjust the interpolation process in order to improve the resulting interpolants. Given a local proof, for example, we can adjust the assignment of inferences to frames [19]. This changes the flow of premises across frame boundaries, which can in turn affect the vocabulary or complexity of the interpolant. Given a non-local proof, we can, for example, adjust the rules used for interpolant generation [14]. This affects the propositional strength of the interpolant produced. The flexibility obtained in these ways is somewhat limited however. For example, in Fig. 6, if we obtained the interpolant sequence $\{\top\}, \{y = 0\}, \{y = 1\}, \{y = 2\}, \{y = 1\}, \{y = 0\}, \{\bot\}$, these methods would not allow us to obtain the more parsimonious sequence using $x = y$. For this a fundamentally different proof is required.

14.6 Abstractors as Proof Generalizers

In our model, the abstractor can be viewed as constructing general proofs from the raw material of the proofs of special cases. We will briefly consider some of the strategies used in the literature for this purpose.

Abstractors fall generally into two categories. The first and most common uses standard fixed-point computation methods (typically either Symbolic Model Checking of an abstract system or explicit-state predicate abstraction) to construct the strongest inductive invariant that can be expressed using a vocabulary provided by the refiner. Example systems in this category include SLAM, BLAST, SATABS [11], Yogi [35], and many others. These abstractors provide a large space of possible proofs. For example, in predicate abstraction, the space of possible safety invariants in L is of size $O(2^{2^N})$, where N is the number of atomic predicates occurring in the refinement proofs. For this reason, various systems either weaken the

proof system (for example, using a Cartesian abstraction) or localize refinement of the proof system to particular paths in the reachability tree or particular abstract states (see Lazy Abstraction [18] and Synergy [17]).

A weaker but less costly alternative to this approach constructs invariants from interpolants in a more direct fashion. Usually this is as either a conjunction or a disjunction of interpolants obtained from special cases. One advantage of this approach is that it avoids the expensive image computation in predicate abstraction, which may involve an exponential number of calls to a decision procedure.

The simplest direct method would be to unwind a loop k times, and to test whether any formula in the sequence interpolant is inductive. This approach is taken, for example in [39]. Slightly more sophisticated would be to find the strongest conjunction or weakest disjunction of the interpolants that is inductive. This is a fixed-point computation, but note that it is in a much smaller space than the one obtained in predicate abstraction (of size 2^k rather than $2^{2^{|P|}}$). Thus, the number of fixed-point iterations is linear instead of exponential in the worst case, and each iteration involves just a linear number of decision procedure calls.

On the other hand, the space of invariants which can be constructed in this way is less rich. For this reason, direct methods generally weaken the refinement paths in some way in order to encourage the refinement prover to generalize. A good example of this is the finite-state Interpolant-Based Model Checking (IBMC) method of [28]. In this method (the first to explicitly use the notion of interpolation in model checking) a path of k iterations is used. A new path is then constructed, weakening the initial condition by replacing it with the previous one-step interpolant.

As mentioned above, weakening of the hypotheses is one method of inducing generalization. It is hoped that some small number of iterations of this process will result in irrelevant information about the initial state being abstracted away, yielding an inductive invariant. On the other hand, over-weakening may result in a failed proof. It can be shown however, that in the finite-state case, a sufficiently large value of k will prevent this. This method provides a complete approach to finite-state model checking using only a SAT solver.

An alternative direct approach is Lazy Abstraction With Interpolants (LAWI) in which the paths in the reachability tree are labeled only with interpolants, and no predicate post-image step is performed [30]. In this technique, a weakening method called *forced covering* is used to encourage generalization.

Finally, abstractors differ in the class of failure cases they produce. In place of "path" in the above discussion, we could have used other sorts of program fragments. For example, we might consider program paths restricted with additional guards, or program fragments containing loops or procedure calls. The choice of failure cases can impact the generality of refinements. As might be expected, more specific cases result in less general refinements, but on the other hand may reduce the cost of searching for a refinement proof.

14.7 Summary

Abstraction means eliminating information that is not relevant to a particular task, such as checking a temporal property of a system model. Here, we have viewed an abstraction as a limited or incomplete proof system. By limiting the proof system, we hope to reduce the proof effort by focusing on relevant facts. When the abstraction fails, we take an approach of generalizing from particular cases to refine it.

In this framework, providing suitable proofs of special cases is the responsibility of the refiner. We have identified two key criteria for refinement proofs. First, they must be expressed in a suitable form to be generalized by the abstractor, a criterion we named *utility*. This means that we must have some effective way to augment the abstractor's proof system to allow it to replicate the refiner's proof. For path-reductive abstractors, we saw that a sufficient condition for utility is a feasible interpolation result for the refiner's proof system. We observed that local proof systems always support feasible interpolation, and moreover that a number of refiners from the literature can in fact be viewed as local proof systems. We can also consider utility in terms of the cost of replicating a refinement proof in the abstractor. For example, refiners such as SP and WP that unnecessarily introduce quantifiers may be considered to have lower utility.

The other important criterion for refinement proofs is the ability to abstract away irrelevant details of special cases. We noted that, according to Occam's razor, simpler proofs are more likely to generalize. Here, we observed a fundamental trade-off. That is, a richer proof system is more likely to allow simple proofs, but on the other hand entails a greater cost in proof search. The various refiners in the literature make this trade-off in different ways.

Another important trade-off is in the space of proofs that the abstractor can construct from the raw material provided by the refiner. Predicate abstraction provides a large space of generalizations, and thus may converge with fewer refinements when compared to IBMC or LAWI. However, this richness comes at a high computational cost, thus predicate abstraction is often weakened. On the other hand, methods that use smaller abstraction spaces generally must put more effort into forcing generality of the refinement proofs.

In a broader sense, this view of "proofs in the aid of model checking" also provides us with some insight into the question "What good is a proof?" We can view a proof as a certificate of correctness. However, in an engineering sense, such a certificate is of questionable value, since at best it guarantees absence of failures that can be captured with a particular model and specification of a system. It can be argued that only "falsification" is of practical interest, since only the discovery of errors leads to actual design changes. Here, however, we have a different view of proof: when searching for errors, a proof guides us away from parts of the space where the errors are not. Conversely, the search for errors directs our proof effort. From a practical point of view, proof and disproof are two sides of the same coin. Strength in one requires strength in the other.

References

1. Ball, T., Majumdar, R., Millstein, T., Rajamani, S.K.: Automatic predicate abstraction of C programs. SIGPLAN Not. **36**(5), 203–213 (2001)
2. Ball, T., Podelski, A., Rajamani, S.K.: Boolean and Cartesian abstraction for model checking C programs. Int. J. Softw. Tools Technol. Transf. **5**(1), 49–58 (2003)
3. Ball, T., Rajamani, S.K.: Bebop: a symbolic model checker for boolean programs. In: Havelund, K., Penix, J., Visser, W. (eds.) Intl. Workshop on Model Checking Software (SPIN). LNCS, vol. 1885, pp. 113–130. Springer, Heidelberg (2000)
4. Ball, T., Rajamani, S.K.: Generating abstract explanations of spurious counterexamples in C programs. Tech. Rep. MSR-TR-2002-09, Microsoft Research (2002)
5. Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 825–885. IOS Press, Amsterdam (2009)
6. Biere, A., Kroening, D.: SAT-based model checking. In: Handbook of Model Checking. Springer, Heidelberg (2018)
7. Brillout, A., Kroening, D., Rümmer, P., Wahl, T.: An interpolating sequent calculus for quantifier-free Presburger arithmetic. In: Giesl, J., Hähnle, R. (eds.) Intl. Joint Conf. on Automated Reasoning (IJCAR). LNCS, vol. 6173, pp. 384–399. Springer, Heidelberg (2010)
8. Brillout, A., Kroening, D., Rümmer, P., Wahl, T.: Beyond quantifier-free interpolation in extensions of Presburger arithmetic. In: Jhala, R., Schmidt, D.A. (eds.) Intl. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI). LNCS, vol. 6538, pp. 88–102. Springer, Heidelberg (2011)
9. Bruttomesso, R., Ghilardi, S., Ranise, S.: Rewriting-based quantifier-free interpolation for a theory of arrays. In: Schmidt-Schauß, M. (ed.) Intl. Conf. Rewriting Techniques and Applications (RTA). LIPIcs, vol. 10, pp. 171–186. Schloss Dagstuhl—Leibniz-Zentrum fuer Informatik, Wadern (2011)
10. Cimatti, A., Griggio, A., Sebastiani, R.: Interpolant generation for UTVPI. In: Schmidt, R.A. (ed.) Intl. Conf. on Automated Deduction (CADE). LNCS, vol. 5663, pp. 167–182. Springer, Heidelberg (2009)
11. Clarke, E.M., Kroening, D., Sharygina, N., Yorav, K.: SATABS: SAT-based predicate abstraction for ANSI-C. In: Halbwachs, N., Zuck, L.D. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 3440, pp. 570–574. Springer, Heidelberg (2005)
12. Craig, W.: Linear reasoning: a new form of the Herbrand-Gentzen theorem. J. Symb. Log. **22**(3), 250–268 (1957)
13. Dams, D., Grumberg, O.: Abstraction and abstraction refinement. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
14. D'Silva, V., Kroening, D., Purandare, M., Weissenbacher, G.: Interpolant strength. In: Barthe, G., Hermenegildo, M.V. (eds.) Intl. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI). LNCS, vol. 5944, pp. 129–145. Springer, Heidelberg (2010)
15. Griggio, A.: Effective word-level interpolation for software verification. In: Bjesse, P., Slobodová, A. (eds.) Formal Methods in Computer Aided Design (FMCAD), pp. 28–36. FMCAD, Austin (2011)
16. Griggio, A., Le, T.T.H., Sebastiani, R.: Efficient interpolant generation in satisfiability modulo linear integer arithmetic. In: Abdulla, P.A., Leino, K.R.M. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 6605, pp. 143–157. Springer, Heidelberg (2011)
17. Gulavani, B.S., Henzinger, T.A., Kannan, Y., Nori, A.V., Rajamani, S.K.: SYNERGY: a new algorithm for property checking. In: Young, M., Devanbu, P.T. (eds.) Intl. Symp. on Foundations of Software Engineering (FSE), pp. 117–127. ACM, New York (2006)
18. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: Jones, N.D., Leroy, X. (eds.) Symp. on Principles of Programming Languages (POPL), pp. 232–244. ACM, New York (2004)

19. Hoder, K., Kovács, L., Voronkov, A.: Playing in the grey area of proofs. In: Field, J., Hicks, M. (eds.) Symp. on Principles of Programming Languages (POPL), pp. 259–272. ACM, New York (2012)
20. Jain, H., Clarke, E.M., Grumberg, O.: Efficient Craig interpolation for linear diophantine (dis)equations and linear modular equations. In: Gupta, A., Malik, S. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 5123, pp. 254–267. Springer, Heidelberg (2008)
21. Jhala, R., McMillan, K.L.: A practical and complete approach to predicate refinement. In: Hermanns, H., Palsberg, J. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 3920, pp. 459–473. Springer, Heidelberg (2006)
22. Jhala, R., Podelski, A., Rybalchenko, A.: Predicate abstraction for program verification. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
23. Kapur, D., Majumdar, R., Zarba, C.G.: Interpolation for data structures. In: Young, M., Devanbu, P.T. (eds.) Intl. Symp. on Foundations of Software Engineering (FSE), pp. 105–116. ACM, New York (2006)
24. Kovács, L., Voronkov, A.: Interpolation and symbol elimination. In: Schmidt, R.A. (ed.) Intl. Conf. on Automated Deduction (CADE). LNCS, vol. 5663, pp. 199–213. Springer, Heidelberg (2009)
25. Kroening, D., Leroux, J., Rümmer, P.: Interpolating quantifier-free Presburger arithmetic. In: Fermüller, C.G., Voronkov, A. (eds.) Intl. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR). LNCS, vol. 6397, pp. 489–503. Springer, Heidelberg (2010)
26. Kroening, D., Sharygina, N.: Interactive presentation: image computation and predicate refinement for RTL verilog using word level proofs. In: Lauwereins, R., Madsen, J. (eds.) Design, Automation & Test in Europe (DATE), pp. 1325–1330. ACM, New York (2007)
27. Kroening, D., Weissenbacher, G.: Lifting propositional interpolants to the word-level. In: Formal Methods in Computer Aided Design (FMCAD), pp. 85–89. IEEE, Piscataway (2007)
28. McMillan, K.L.: Interpolation and SAT-based model checking. In: Hunt, W.A., Somenzi, F. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 2725, pp. 1–13. Springer, Heidelberg (2003)
29. McMillan, K.L.: An interpolating theorem prover. Theor. Comput. Sci. **345**(1), 101–121 (2005)
30. McMillan, K.L.: Lazy abstraction with interpolants. In: Ball, T., Jones, R.B. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 4144, pp. 123–136. Springer, Heidelberg (2006)
31. McMillan, K.L.: Quantified invariant generation using an interpolating saturation prover. In: Ramakrishnan, C.R., Rehof, J. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 4963, pp. 413–427. Springer, Heidelberg (2008)
32. McMillan, K.L.: Lazy annotation for program testing and verification. In: Touili, T., Cook, B., Jackson, P. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 6174, pp. 104–118. Springer, Heidelberg (2010)
33. McMillan, K.L.: Interpolants from Z3 proofs. In: Bjesse, P., Slobodová, A. (eds.) Formal Methods in Computer Aided Design (FMCAD), pp. 19–27. FMCAD, Austin (2011)
34. McMillan, K.L., Amla, N.: Automatic abstraction without counterexamples. In: Garavel, H., Hatcliff, J. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 2619, pp. 2–17. Springer, Heidelberg (2003)
35. Nori, A.V., Rajamani, S.K., Tetali, S., Thakur, A.V.: The Yogi project: software property checking via static analysis and testing. In: Kowalewski, S., Philippou, A. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 5505, pp. 178–181. Springer, Heidelberg (2009)
36. Pudlák, P.: Lower bounds for resolution and cutting plane proofs and monotone computations. J. Symb. Log. **62**(3), 981–998 (1997)

37. Rybalchenko, A., Sofronie-Stokkermans, V.: Constraint solving for interpolation. J. Symb. Comput. **45**(11), 1212–1233 (2010)
38. Shankar, N.: Combining model checking and deduction. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
39. Vizel, Y., Grumberg, O.: Interpolation-sequence based model checking. In: Formal Methods in Computer Aided Design (FMCAD), pp. 1–8. IEEE, Piscataway (2009)
40. Yorsh, G., Musuvathi, M.: A combination method for generating interpolants. In: Nieuwenhuis, R. (ed.) Intl. Conf. on Automated Deduction (CADE). LNCS, vol. 3632, pp. 353–368. Springer, Heidelberg (2005)

Chapter 15
Predicate Abstraction for Program Verification

Safety and Termination

Ranjit Jhala, Andreas Podelski, and Andrey Rybalchenko

Abstract We present basic principles of algorithms for the verification of safety and termination of programs. The algorithms call procedures on logical formulas in order to construct an abstraction and to refine an abstraction. The two underlying concepts are predicate abstraction and counterexample-guided abstraction refinement.

15.1 Introduction

In this chapter, we are interested in program verification algorithms, i.e., in algorithms that take a program and a correctness property and try to answer the question whether the program is correct. Correctness is expressed by one of two properties of program executions: *safety* (which we formalize as the non-reachability of given *error* states), and *termination*. We are interested in a general class of programs for which safety and termination are not decidable. As a consequence, the algorithms must be based on abstraction.

The distinguishing feature of the algorithms is a specific way to call procedures over logical formulas in order to effectively construct an abstraction and to effectively refine this abstraction. The two underlying concepts are predicate abstraction and counterexample-guided abstraction refinement.

An abstraction maps a set of states to a superset. The terminology *predicate abstraction* refers to the fact that the superset is constructed from a basis of so-called *predicates* (pre-selected formulas that define sets of states). Now, with more predicates one has a larger choice for the construction of the superset, and the abstraction

R. Jhala
Jacobs School of Engineering, University of California, San Diego, San Diego, CA, USA

A. Podelski
University of Freiburg, Freiburg, Germany

A. Rybalchenko (✉)
Microsoft Research, Cambridge, UK
e-mail: rybal@microsoft.com

E.M. Clarke et al. (eds.), *Handbook of Model Checking*,
DOI 10.1007/978-3-319-10575-8_15

can be more precise. In this sense, adding more predicates refines the abstraction. The terminology *abstraction refinement* refers to the process of adding new predicates. The crux of the verification algorithms is the *counterexample-guided* procedure to select new predicates.

In an analogous way, we use *transition predicates* in order to construct the abstraction of a transition relation (a set of pairs of states).

Program verification with predicate abstraction is an ongoing research topic. We can expect a great number of variations and optimizations to be proposed in the future. Yet, a few basic principles have emerged which will remain the basis for further developments even in the long term. Those few basic principles keep reappearing in different settings, each setting being motivated by a specific application scenario. The idea of this chapter is to abstract away from specific application scenarios and to present the few basic principles in the shortest possible way in the simplest possible formalism. For an exposition of the wealth of existing work in this area we refer to the survey in [42]. An account of the history of counterexample-guided abstraction refinement is given in [19].

15.2 Definitions

In this section, we use a formal setting based on logical formulas in order to introduce programs, computations, and two representative properties of computations, namely safety and termination.

15.2.1 Programs

We specify a program formally through logical formulas. For an example, see Fig. 1.

We assume a set V of logical variables that we call *program variables*. Each program variable comes with a *domain* (a set of values, e.g., integers).

The program counter *pc* is a distinguished program variable of every program, i.e., $pc \in V$. The domain of the program counter is a (finite) set *Loc* of special values called the *control locations* of the program.

A *program state* s is a function that assigns each program variable a value from its respective domain. Let Σ be the set of program states.

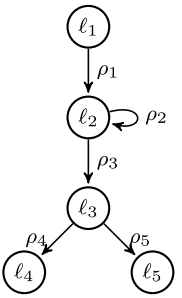
We sometimes fix an order on the variables by writing V as a tuple of variables, say $V = (pc, x, y, z)$, and then use a tuple of values to denote a state, e.g., $s = (\ell_1, 1, 3, 2)$.

A formula φ with free variables in V represents a set of program states. For example, the formula $pc = \ell$ represents the set of all states at the control location ℓ. The formula $x > 0$ represents the set of states (at any program location) where the program variable x has a value strictly greater than 0.

Each program variable (including *pc*) comes with its primed version. That is, for each program variable x in V, we have another variable x'. We write V' for the tuple

```
main(int x, int y, int z) {
  assume(y >= z);
  while (x < y) {
    x++;
  }
  assert(x >= z);
}
```

(a)



(b)

$$\rho_1 = (goto(\ell_1, \ell_2) \wedge y \geq z \wedge unchanged(x, y, z))$$
$$\rho_2 = (goto(\ell_2, \ell_2) \wedge x + 1 \leq y \wedge x' = x + 1 \wedge unchanged(y, z))$$
$$\rho_3 = (goto(\ell_2, \ell_3) \wedge x \geq y \wedge unchanged(x, y, z))$$
$$\rho_4 = (goto(\ell_3, \ell_4) \wedge x \geq z \wedge unchanged(x, y, z))$$
$$\rho_5 = (goto(\ell_3, \ell_5) \wedge x + 1 \leq z \wedge unchanged(x, y, z))$$

(c)

Fig. 1 An example program (**a**), its control flow graph (**b**), and its transition relations (**c**). Formally, the program is $\mathcal{P} = (V, pc, \varphi_{init}, \mathcal{T}, \varphi_{err})$ where $V = (pc, x, y, z)$ is the tuple of program variables, pc is the program counter variable, $\mathcal{T} = \{\rho_1, \rho_2, \rho_3, \rho_4, \rho_5\}$ is the set of transition relations, $\varphi_{init} = at\_\ell_1$ is the initial condition, and $\varphi_{err} = at\_\ell_5$ is the error condition. The primed variables are $V' = (pc', x', y', z')$. We use *goto* and *unchanged* as abbreviations. For example $goto(\ell_1, \ell_2)$ stands for $(pc = \ell_1 \wedge pc' = \ell_2)$ and $unchanged(x, y, z)$ stands for $(x' = x \wedge y' = y \wedge z' = z)$

of primed versions of program variables. A formula ψ with free variables in V and V' represents a set of pairs of states, i.e., a binary relation over states.

Formally, the pair (s_1, s_2) defines a valuation ν of variables in $V \cup V'$ where $\nu(x) = s_1(x)$ and $\nu(x') = s_2(x)$ for each variable x in V (and thus x' in V'). A formula ψ in unprimed and primed variables represents the set of pairs of states (s_1, s_2) such that the corresponding valuation ν satisfies ψ.

For example, the formula $pc = \ell_1 \wedge pc' = \ell_2$ represents the set of pairs of states (s_1, s_2) whose first component s_1 is a state at the control location ℓ_1 and whose second component s_2 is a state at the control location ℓ_2. The formula $x' = x$ represents the set of pairs of states (s_1, s_2) (at any program location) where the program variable x has the same value in the state s_1 and in the state s_2. The formula $x > 0 \wedge x' > x$ represents the set of pairs of states (s_1, s_2) where the program variable x has a value greater than 0 in the state s_1 and its value in the state s_1 is smaller than in the state s_2.

The formula $x' > 0$ represents the set of pairs of states (s_1, s_2) where the program variable x has a value greater than 0 in the state s_2 and its value in the state s_1 is unconstrained. Symmetrically, the formula $x > 0$ represents the set of pairs of states

(s_1, s_2) where the program variable x has a value greater than 0 in the state s_1 and its value in the state s_2 is unconstrained.

We can use a formula φ in unprimed variables to represent both a set of states and a binary relation over states. Thus, we can represent the restriction of the binary relation ψ to the set φ by the conjunction $\psi \wedge \varphi$.

To simplify the notation for transition relations, we introduce the following abbreviations (here ℓ, ℓ_1, and ℓ_2 are control locations and $x_1, \ldots, x_n$ are program variables).

$$\begin{aligned} at\_\ell &= (pc = \ell), \\ at'\_\ell &= \big(pc' = \ell\big), \\ goto(\ell_1, \ell_2) &= \big(at\_\ell_1 \wedge at'\_\ell_2\big), \\ unchanged(x_1, \ldots, x_n) &= \big(x_1' = x_1 \wedge \cdots \wedge x_n' = x_n\big). \end{aligned} \tag{1}$$

A program $\mathcal{P}$ is specified by the tuple $\mathcal{P} = (V, pc, \varphi_{init}, \mathcal{T}, \varphi_{err})$ consisting of the set of program variables V, the program counter pc, the initiation condition φ_{init}, the set of transition relations $\mathcal{T} = \{\rho_1, \ldots, \rho_n\}$, and the error condition φ_{err}.

The *initiation condition* φ_{init} and the *error condition* φ_{err} are formulas over variables in V. They represent the set of *initial states* and the set of *error states*, respectively.

The elements $\rho_1, \ldots, \rho_n$ are formulas over the program variables in V and their primed versions V'. If the formula ρ_i contains a conjunct of the form $goto(\ell_1, \ell_2)$ for two locations ℓ_1 and ℓ_2, we say that ρ_i is a *transition* from ℓ_1 to ℓ_2.

The set of transition relations $\mathcal{T} = \{\rho_1, \ldots, \rho_n\}$ defines the *program transition relation* of $\mathcal{P}$, which is represented by the formula

$$\rho_{\mathcal{P}} = \rho_1 \vee \cdots \vee \rho_n \ . \tag{2}$$

The formula $\rho_{\mathcal{P}}$ thus represents the union of the transition relations represented by the transitions $\rho_1, \ldots, \rho_n$.

Example 1 Our example program has an initiation condition $\varphi_{init} = (at\_\ell_1)$ and an error condition $\varphi_{err} = (at\_\ell_5)$. That is, every state at control location ℓ_1 is an initial state and every state at control location ℓ_5 is an error state. The set of program transitions $\mathcal{T} = \{\rho_1, \rho_2, \rho_3, \rho_4, \rho_5\}$ corresponds to the graph as shown in Fig. 1(b). We call this graph the *control flow graph* of the program. The transition relations ρ_1, ρ_2, ρ_3, ρ_4, and ρ_5 are defined in Fig. 1(c). The transition relation of the program is the disjunction $\rho_{\mathcal{P}} = \rho_1 \vee \rho_2 \vee \rho_3 \vee \rho_4 \vee \rho_5$. □

It is convenient to identify formulas with the sets and relations that they represent. Accordingly, we identify the logical consequence relation (entailment) between formulas $\models$ with the set inclusion relation $\subseteq$ between the sets that they represent. (All examples presented in this chapter use the theory of linear rational arithmetic.) Furthermore, we identify the satisfaction relation between a valuation and a formula

(which is also denoted by $\models$) with the membership relation $\in$ between the corresponding state and the represented set of states (or between the corresponding pair of states and the represented relation between states).

Often, a formal setting for program verification is based on the notion of a *control flow graph*, i.e., a graph whose nodes correspond to the program locations and whose edges are labeled by statements. This may reflect a particular design decision in a practical implementation. It is clear, however, that one can derive the logical formula denoting the transition relation of the program from a control flow graph, and vice versa. As in the example, the logical formula denoting the transition relation of the program induces a graph where, e.g., each edge (ℓ_1, ℓ_2) arises from a conjunct $goto(\ell_1, \ell_2)$ in the logical formula. By starting directly with logical formulas, we obtain a uniform setting.

Example 2 Consider the program shown in Fig. 1. Let s be the program state given by the tuple $(\ell_1, 1, 3, 2)$ (which stands for the mapping that assigns 1, 3, 2, and ℓ_1 to the program variables x, y, z, and pc, respectively). Then, we have $s \models y \geq z$ (or, written differently, $s \in y \geq z$). Furthermore, we have $y \geq z \models y + 1 \geq z$ (or, written differently, $y \geq z \subseteq y + 1 \geq z$). □

15.2.2 Correctness: Safety and Termination

Given a program $\mathcal{P}$ with the program transition relation $\rho_{\mathcal{P}}$, the set of initial states φ_{init}, and the set of error states φ_{err}, we formalize program correctness as a property of program computations. A *program computation* of $\mathcal{P}$ is either a finite sequence $s_1, \ldots, s_n$ or an infinite sequence $s_1, s_2, \ldots$ of states that is generated by the program transition relation $\rho_{\mathcal{P}}$, starts in an initial state, and if it is finite then it cannot be continued after the last state (s_n is a *deadlock* state). This means:

- each pair of consecutive states s_i and s_{i+1} in the sequence is an element of the program transition relation, i.e., $(s_i, s_{i+1}) \in \rho_{\mathcal{P}}$,
- the first element of the sequence is an initial state, i.e., $s_1 \in \varphi_{init}$,
- if the sequence is finite with s_n as its last element, then the state s_n does not have any successor state w.r.t. the program transition relation $\rho_{\mathcal{P}}$, i.e., there is no state s such that $(s_n, s) \in \rho_{\mathcal{P}}$.

We will write $s_1, s_2, \ldots$ for program computations, whether finite or infinite.

Example 3 The (finite) sequence of states below is a program computation in our example program $\mathcal{P}$.

$$(\ell_1, 1, 3, 2), (\ell_2, 1, 3, 2), (\ell_2, 2, 3, 2), (\ell_2, 3, 3, 2), (\ell_3, 3, 3, 2), (\ell_4, 3, 3, 2)$$

The sequence of states starts in an initial state and follows the sequence of transitions $\rho_1, \rho_2, \rho_2, \rho_3, \rho_4$. The last state in the sequence does not have any successor state w.r.t. the program transition relation $\rho_{\mathcal{P}}$. □

The verification of a large class of properties of program computations can be reduced to reasoning about safety and termination.

A program is *safe* if no error state occurs in any program computation. A program *terminates* if every program computation is finite.

A finite-state program terminates if and only if the length of program computations is bounded. In general, the length of program computations is unbounded even if the program is terminating (see, for example, the program in Fig. 1).

15.3 Characterizing Correctness via Reachability

We will next characterize safety and termination by conditions that are suitable for the abstraction-based verification of safety and termination. The conditions are defined in terms of reachability of states and, respectively, reachability of pairs of states (binary reachability).

15.3.1 Safety and Reachability

A state s is *reachable* if there exists a program computation $s_1, s_2, \ldots$ with an occurrence of s (i.e., there exists a position i such that $s_i = s$). We use

$$\varphi_{reach}$$

for the set of all reachable states.

An *invariant* is a set φ that contains all reachable states, i.e., $\varphi_{reach} \subseteq \varphi$.

The program $\mathcal{P}$ is *safe* if and only if the complement of the set of error states is an invariant, i.e., if

$$\varphi_{reach} \subseteq \Sigma \setminus \varphi_{err}. \tag{3}$$

Example 4 For our example program, the set of reachable states is shown below.

$$\varphi_{reach} = \big(at\_\ell_1 \vee (at\_\ell_2 \wedge y \geq z) \\ \vee (at\_\ell_3 \wedge y \geq z \wedge x \geq y) \vee (at\_\ell_4 \wedge y \geq z \wedge x \geq y)\big).$$

This set does not contain any error states, i.e., we have

$$\varphi_{reach} \subseteq \neg at\_\ell_5. \qquad \square$$

In Sect. 15.4.1, we show that one can construct the set φ_{reach} by an iterative application of a function on sets of states. In general, one needs to iterate the application of the function infinitely many times. In Sect. 15.5.1, we show that one can construct a superset of φ_{reach} by an iterative application of an abstraction of the function. We construct the abstract function automatically using *predicate abstraction*. With predicate abstraction, one needs to iterate the application of the abstract function only finitely many times.

15.3.2 Termination and Binary Reachability

We extend the notion of reachability from states to pairs of states. A pair of states (s, s') is *reachable* if s is reachable from the initial state and s' is reachable from s, that is, if there exists a program computation in which s is followed by s' (i.e., the program computation is of the form $s_1, \ldots, s_i, \ldots, s_j, \ldots$ where $s_i = s$ and $s_j = s'$ for positions i and j such that $1 \leq i < j$). We use

$$\psi_{reach}$$

for the set of reachable pairs of states and call it the *binary reachability* relation.

A *transition invariant* is a binary relation over states ψ that contains the binary reachability relation, i.e., $\psi_{reach} \subseteq \psi$.

Just as we used the notion of invariant to characterize safety, we will use the notion of a transition invariant to characterize termination. The interest of the characterization of termination in this way lies in a proof method for termination which parallels the proof method for safety. As we show in Sect. 15.4.2, one can construct the set ψ_{reach} by an iterative application of a function on sets of pairs of states. In general, one needs to iterate the application of the function infinitely many times. In Sect. 15.5.2, we show that one can construct a superset of ψ_{reach} by an iterative application of an abstraction of the function. We construct the abstract function automatically using the analogue of predicate abstraction for *transition predicate*s. One needs to iterate the application of the abstract function only finitely many times.

The termination of a program can be equivalently expressed as the well-foundedness of its program transition relation. A binary relation ψ is defined to be *well-founded* if it does not generate any infinite sequence (i.e., if there is no infinite sequence $s_1, s_2, \ldots$ such that $(s_i, s_{i+1}) \in \psi$ for all $i = 1, 2, \ldots$). For example, the relation $x > 0 \wedge x' < x$ is well-founded. The union of well-founded relations is in general not well-founded (take, for example, the union of the relations $x > 0 \wedge x' < x$ and $y > 0 \wedge y' < y$).

Assume we are given a number of well-founded relations $\psi_1, \ldots, \psi_n$ (each corresponding, for example, to the program transition relation of a terminating program). The program $\mathcal{P}$ is terminating if the union of the n well-founded relations, which we call a disjunctively well-founded relation, is a transition invariant, i.e., if

$$\psi_{reach} \subseteq \psi_1 \cup \cdots \cup \psi_n. \tag{4}$$

The proof of this fact relies on Ramsey's theorem on combinatorics for infinite graphs, see [52].

Just as we used the notion of invariant to characterize safety, we have used the notion of a transition invariant to characterize termination. The interest of the characterization of termination by transition invariants lies in a proof method for termination which parallels the proof method for safety. As we show in Sect. 15.4.2, one can construct the set ψ_{reach} by an iterative application of a function on sets of pairs of states. In general, one needs to iterate the application of the function infinitely many times. In Sect. 15.5.2, we show that one can construct a superset of

ψ_{reach} by an iterative application of an abstraction of the function. We construct the abstract function automatically using the analogue of predicate abstraction for *transition predicate*s. One needs to iterate the application of the abstract function only finitely many times.

To be precise, we have characterized termination by the fact that the union of a (finite) number of well-founded relations forms a transition invariant. We have not said where the well-founded relations come from. For the purpose of this presentation, we assume that they are given. There are, however, many strategies to obtain formulas that represent the required well-founded relations; see, e.g., [20, 52].

15.4 Characterizing Correctness via Inductiveness

In order to check reachability (or binary reachability), we need to construct the set of reachable states (or the set of reachable pairs of states). The construction is possible, in theory, by the iterative application of a function over sets of states (or a function over sets of pairs of states). This construction may need infinitely many iterations. It defines the smallest set that is *inductive*, i.e., closed under the application of the function. We may not need to construct the smallest set; it may be sufficient to construct a superset. In order to show that a given set is indeed a superset of the set of reachable states (or the set of reachable pairs of states), it is sufficient to show that it is inductive.

15.4.1 Safety and Closure Under post

Let φ be a formula over V and let ρ be a formula over V and V'. We define a *post-condition* function *post* as follows.

$$post(\varphi, \rho) = \exists V'' : \varphi\big[V''/V\big] \wedge \rho\big[V''/V\big]\big[V/V'\big]. \tag{5}$$

Here $\varphi[V''/V]$ represents the result of replacing V by V'' in φ, while $\rho[V''/V]$ $[V/V']$ requires first replacing V by V'' and then replacing V' by V. An application $post(\varphi, \rho)$ computes the image of the set φ under the relation ρ. We observe the following useful property of the post-condition function.

$$\begin{aligned} &\forall\varphi\ \forall\rho_1\ \forall\rho_2 : post(\varphi, \rho_1 \vee \rho_2) = \big(post(\varphi, \rho_1) \vee post(\varphi, \rho_2)\big); \\ &\forall\varphi_1\ \forall\varphi_2\ \forall\rho : post(\varphi_1 \vee \varphi_2, \rho) = \big(post(\varphi_1, \rho) \vee post(\varphi_2, \rho)\big). \end{aligned} \tag{6}$$

This property states that the post-condition computation distributes over disjunction w.r.t. each argument.

Furthermore, for a natural number n we define $post^n(\varphi, \rho)$ to represent the n-fold application of the *post* function to φ with respect to ρ. Formally, we have:

$$post^n(\varphi, \rho) = \begin{cases} \varphi & \text{if } n = 0 \\ post(post^{n-1}(\varphi, \rho), \rho) & \text{otherwise.} \end{cases} \tag{7}$$

Example 5 Given the transition relation ρ_2 and the program variables $V = (pc, x, y, z)$ from our example program, we compute the following post-condition.

$$\begin{aligned}
&post(at_\ell_2 \wedge y \geq z, \rho_2) \\
&\quad = \big(\exists V'' : (at_\ell_2 \wedge y \geq z)\big[V''/V\big] \wedge \rho_2\big[V''/V\big]\big[V/V'\big]\big) \\
&\quad = \big(\exists V'' : \big(pc'' = \ell_2 \wedge y'' \geq z''\big) \\
&\qquad \wedge \big(pc'' = \ell_2 \wedge pc' = \ell_2 \wedge x'' + 1 \leq y'' \wedge x' = x'' + 1 \\
&\qquad \wedge y' = y'' \wedge z' = z''\big)\big[V/V'\big]\big) \\
&\quad = \big(\exists V'' : \big(pc'' = \ell_2 \wedge y'' \geq z''\big) \\
&\qquad \wedge \big(pc'' = \ell_2 \wedge pc = \ell_2 \wedge x'' + 1 \leq y'' \wedge x = x'' + 1 \\
&\qquad \wedge y = y'' \wedge z = z''\big)\big) \\
&\quad = (pc = \ell_2 \wedge y \geq z \wedge x \leq y).
\end{aligned}$$

We compute the 2-fold application by reusing the above result.

$$\begin{aligned}
&post^2(at_\ell_2 \wedge y \geq z, \rho_2) \\
&\quad = post\big(post(at_\ell_2 \wedge y \geq z, \rho_2), \rho_2\big) \\
&\quad = post(pc = \ell_2 \wedge y \geq z \wedge x \leq y, \rho_2) \\
&\quad = \big(\exists V'' : \big(pc'' = \ell_2 \wedge y'' \geq z'' \wedge x'' \leq y''\big) \\
&\qquad \wedge \big(pc'' = \ell_2 \wedge pc = \ell_2 \wedge x'' + 1 \leq y'' \wedge x = x'' + 1 \\
&\qquad \wedge y = y'' \wedge z = z''\big)\big) \\
&\quad = (pc = \ell_2 \wedge y \geq z \wedge x - 1 \leq y \wedge x \leq y) \\
&\quad = (pc = \ell_2 \wedge y \geq z \wedge x \leq y).
\end{aligned}$$

□

We characterize φ_{reach} using *post* as follows.

$$\begin{aligned}
\varphi_{reach} &= \varphi_{init} \vee post(\varphi_{init}, \rho_{\mathcal{P}}) \vee post\big(post(\varphi_{init}, \rho_{\mathcal{P}}), \rho_{\mathcal{P}}\big) \vee \ldots \\
&= \bigvee_{i \geq 0} post^i(\varphi_{init}, \rho_{\mathcal{P}}).
\end{aligned} \tag{8}$$

The above disjunction (over every number of applications of the post-condition function) ensures that all reachable states are taken into consideration.

Example 6 We compute φ_{reach} for our example program. We first obtain the post-condition after applying the transition relation of the program once.

$$\begin{aligned}
&post(at_\ell_1, \rho_{\mathcal{P}}) \\
&\quad = \big(post(at_\ell_1, \rho_1) \vee post(at_\ell_1, \rho_2) \vee post(at_\ell_1, \rho_3) \\
&\qquad \vee post(at_\ell_1, \rho_4) \vee post(at_\ell_1, \rho_5)\big) \\
&\quad = post(at_\ell_1, \rho_1) \\
&\quad = (at_\ell_2 \wedge y \geq z).
\end{aligned}$$

Next, we obtain the post-condition for one more application.

$$\begin{aligned}
&post(at_\ell_2 \wedge y \geq z, \rho_{\mathcal{P}}) \\
&\quad = \big(post(at_\ell_2 \wedge y \geq z, \rho_2) \vee post(at_\ell_2 \wedge y \geq z, \rho_3)\big) \\
&\quad = (at_\ell_2 \wedge y \geq z \wedge x \leq y \vee at_\ell_3 \wedge y \geq z \wedge x \geq y).
\end{aligned}$$

We repeat the application step once again.

$$\begin{aligned}
&post(at_\ell_2 \wedge y \geq z \wedge x \leq y \vee at_\ell_3 \wedge y \geq z \wedge x \geq y, \rho_{\mathcal{P}}) \\
&\quad = \big(post(at_\ell_2 \wedge y \geq z \wedge x \leq y, \rho_{\mathcal{P}}) \vee post(at_\ell_3 \wedge y \geq z \wedge x \geq y, \rho_{\mathcal{P}})\big) \\
&\quad = \big(post(at_\ell_2 \wedge y \geq z \wedge x \leq y, \rho_2) \vee post(at_\ell_2 \wedge y \geq z \wedge x \leq y, \rho_3) \\
&\qquad \vee post(at_\ell_3 \wedge y \geq z \wedge x \geq y, \rho_4) \vee post(at_\ell_3 \wedge y \geq z \wedge x \geq y, \rho_5)\big) \\
&\quad = (at_\ell_2 \wedge y \geq z \wedge x \leq y \vee at_\ell_3 \wedge y \geq z \wedge x = y \\
&\qquad \vee at_\ell_4 \wedge y \geq z \wedge x \geq y).
\end{aligned}$$

So far, by iteratively applying the post-condition function to φ_{init} we obtained the following disjunction.

$$\begin{aligned}
&at_\ell_1 \vee \\
&at_\ell_2 \wedge y \geq z \vee \\
&at_\ell_2 \wedge y \geq z \wedge x \leq y \vee at_\ell_3 \wedge y \geq z \wedge x \geq y \vee \\
&at_\ell_2 \wedge y \geq z \wedge x \leq y \vee at_\ell_3 \wedge y \geq z \wedge x = y \vee \\
&at_\ell_4 \wedge y \geq z \wedge x \geq y.
\end{aligned}$$

We present this disjunction in a logically equivalent, simplified form as follows.

$$
\begin{aligned}
&at\_\ell_1 \vee \\
&at\_\ell_2 \wedge y \geq z \vee \\
&at\_\ell_3 \wedge y \geq z \wedge x \geq y \vee \\
&at\_\ell_4 \wedge y \geq z \wedge x \geq y.
\end{aligned}
$$

Any further application of the post-condition function does not produce any additional disjuncts. Hence, φ_{reach} is the above disjunction. □

Inductive Proof of Safety

An *inductive invariant* φ contains the initial states and is closed under successors [30, 40]. Formally, an inductive invariant is a formula over the program variables that represents a superset of the initial program states and is closed under the application of the *post* function w.r.t. the relation $\rho_{\mathcal{P}}$, i.e.,

$$\varphi_{init} \models \varphi \quad \text{and} \quad post(\varphi, \rho_{\mathcal{P}}) \models \varphi.$$

A program is safe if there exists an inductive invariant φ that does not contain any error states, i.e., $\varphi \wedge \varphi_{err} \models false$.

Example 7 For our example program, the weakest inductive invariant consists of the set of all states and is represented by the formula *true*. The strongest inductive invariant was obtained in Example 6. The strongest inductive invariant does not contain any error states. We observe that the slightly weaker inductive invariant below also proves the safety of our examples.

$$at\_\ell_1 \vee (at\_\ell_2 \wedge y \geq z) \vee (at\_\ell_3 \wedge y \geq z \wedge x \geq y) \vee at\_\ell_4. \qquad \square$$

15.4.2 Termination and Transitive Closure

Let ρ_1 and ρ_2 be formulas over V and V'. We define a *relational composition* function $\circ$ as follows.

$$\rho_1 \circ \rho_2 = \exists V'' : \rho_1\big[V''/V'\big] \wedge \rho_2\big[V''/V\big]. \tag{9}$$

Example 8 Given the transition relations ρ_1, ρ_2, and the program variables $V = (pc, x, y, z)$ from our example program we obtain the following relational composi-

tion.

$$\begin{aligned}
\rho_1 \circ \rho_2 = \big(\exists V'' : \big(&pc = \ell_1 \wedge pc' = \ell_2 \wedge y \geq z \\
&\wedge x' = x \wedge y' = y \wedge z' = z\big)\big[V''/V'\big] \\
&\wedge \big(pc = \ell_2 \wedge pc' = \ell_2 \wedge x + 1 \leq y \\
&\wedge x' = x + 1 \wedge y' = y \wedge z' = z\big)\big[V''/V\big]\big) \\
= \big(\exists V'' : \big(&pc = \ell_1 \wedge pc'' = \ell_2 \wedge y \geq z \\
&\wedge x'' = x \wedge y'' = y \wedge z'' = z\big) \\
&\wedge \big(pc'' = \ell_2 \wedge pc' = \ell_2 \wedge x'' + 1 \leq y'' \\
&\wedge x' = x'' + 1 \wedge y' = y'' \wedge z' = z''\big)\big) \\
= \big(&pc = \ell_1 \wedge pc' = \ell_2 \wedge y \geq z \wedge x + 1 \leq y \\
&\wedge x' = x + 1 \wedge y' = y \wedge z' = z\big).
\end{aligned}$$

□

For a given $\rho_{\mathcal{P}}$, a binary relation ψ and a natural number n, we define the *n-time transition composition* $comp^n(\rho_{\mathcal{P}})$ of $\rho_{\mathcal{P}}$ with ψ as follows.

$$comp^n(\psi) = \begin{cases} \psi & \text{if } n = 0 \\ comp^{n-1}(\psi) \circ \rho_{\mathcal{P}} & \text{otherwise.} \end{cases}$$

We can compute the (irreflexive) transitive closure $\rho_{\mathcal{P}}^+$ using *comp* as follows.

$$\begin{aligned}
\rho_{\mathcal{P}}^+ &= \rho_{\mathcal{P}} \vee \rho_{\mathcal{P}} \circ \rho_{\mathcal{P}} \vee \rho_{\mathcal{P}} \circ \rho_{\mathcal{P}} \circ \rho_{\mathcal{P}} \vee \ldots \\
&= \bigvee_{i \geq 1} comp^i(\rho_{\mathcal{P}}). \qquad (10)
\end{aligned}$$

We will be using a restriction of $\rho_{\mathcal{P}}^+$ to reachable states φ_{reach}. For this reason, we define

$$\psi_{ti} = \bigvee_{i \geq 1} comp^i\big(\varphi_{reach} \wedge V' = V\big). \qquad (11)$$

That is, ψ_{ti} is a transition invariant that is characterized using iteration of relational composition.

Inductive Proof for Termination

The restriction of the program transition relation $\rho_{\mathcal{P}}$ to the reachable program states is given by $\rho_{\mathcal{P}} \wedge \varphi_{reach}$ (the conjunction of a formula over V and V' and a formula over V). A program terminates if and only if the binary relation $\rho_{\mathcal{P}} \wedge \varphi_{reach}$ is well-founded.

Example 9 For our example, we obtain the following restriction of the program transition relation to reachable states.

$$\begin{aligned}
&\rho_{\mathcal{P}} \wedge \varphi_{reach} \\
&\quad = \big(goto(\ell_1, \ell_2) \wedge y \geq z \wedge unchanged(x, y, z) \\
&\qquad \vee goto(\ell_2, \ell_2) \wedge y \geq z \wedge x + 1 \leq y \wedge x' = x + 1 \wedge unchanged(y, z) \\
&\qquad \vee goto(\ell_2, \ell_3) \wedge y \geq z \wedge x \geq y \wedge unchanged(x, y, z) \\
&\qquad \vee goto(\ell_3, \ell_4) \wedge y \geq z \wedge x \geq y \wedge x \geq z \wedge unchanged(x, y, z)\big).
\end{aligned}$$

The restriction consists of four disjuncts, since the transition relation ρ_5 does not intersect with φ_{reach}. Furthermore, the restriction is well-founded, i.e., our program terminates. Any attempt to construct an infinite sequence leads to unbounded increase of the values of the variable x, which contradicts the condition that x is bounded from above by y whenever the loop execution is carried out. □

An *inductive transition invariant* ψ contains the restriction of the program transition relation to reachable states and is closed under relational composition with the program transition relation [52]. Formally, given an inductive invariant φ, we require that an inductive transition invariant ψ satisfies the following conditions:

$$\varphi \wedge \rho_{\mathcal{P}} \models \psi \quad \text{and} \quad \psi \circ \rho_{\mathcal{P}} \models \psi.$$

A program terminates if there exist a finite number of well-founded relations $\psi_1, \ldots, \psi_n$ whose union contains an inductive transition invariant, i.e., $\psi \models \psi_1 \vee \cdots \vee \psi_n$.

15.5 Abstraction

The computation of the set of reachable program states requires the iterative application of the post-condition function on the initial program states, see Eq. (8). The iteration stops when no new disjuncts are being added. Unfortunately, in many cases, the iteration will never stop.

Example 10 We consider the iterative computation of the set of states that is reachable from $at\_\ell_2 \wedge x \leq z$ by applying the transition ρ_2 of our example program. We obtain the following sequence of post-conditions (where $V = (pc, x, y, z)$).

$$\begin{aligned}
post(at_\ell_2 \wedge x \leq z, \rho_2) &= \big(\exists V'' : \big(pc'' = \ell_2 \wedge x'' \leq z''\big) \\
&\quad \wedge \big(pc'' = \ell_2 \wedge pc = \ell_2 \wedge x'' + 1 \leq y'' \\
&\quad \wedge x = x'' + 1 \wedge y = y'' \wedge z = z''\big)\big) \\
&= (at_\ell_2 \wedge x - 1 \leq z \wedge x \leq y)
\end{aligned}$$

$$post^2(at\_\ell_2 \wedge x \leq z, \rho_2) = (at\_\ell_2 \wedge x - 2 \leq z \wedge x \leq y)$$

$$post^3(at\_\ell_2 \wedge x \leq z, \rho_2) = (at\_\ell_2 \wedge x - 3 \leq z \wedge x \leq y)$$

$$\ldots$$

$$post^n(at\_\ell_2 \wedge x \leq z, \rho_2) = (at\_\ell_2 \wedge x - n \leq z \wedge x \leq y).$$

In this sequence, we observe that each iteration yields a set of states that contains states not discovered before. For example, the set of states reachable after applying the post-condition function once is not included in the original set, i.e.,

$$(at\_\ell_2 \wedge x - 1 \leq z \wedge x \leq y) \not\models (at\_\ell_2 \wedge x \leq z).$$

The set of states reachable after applying the post-condition function twice is not included in the union of the above two sets, i.e.,

$$(at\_\ell_2 \wedge x - 2 \leq z \wedge x \leq y) \not\models (at\_\ell_2 \wedge x - 1 \leq z \wedge x \leq y \vee at\_\ell_2 \wedge x \leq z).$$

Furthermore, we observe that the set of states reachable after n-fold application of *post*, where $n \geq 1$, still contains previously unreached states, i.e.,

$$\forall n \geq 1 : (at\_\ell_2 \wedge x - n \leq z \wedge x \leq y) \not\models \left(at\_\ell_2 \wedge x \leq z \vee \bigvee_{1 \leq i < n} (at\_\ell_2 \wedge x - i \leq z \wedge x \leq y) \right).$$

□

A similar example can be used to show the possibility of non-termination for the procedure which constructs the strongest transition invariant.

15.5.1 Safety and Predicate Abstraction

Instead of computing φ_{reach}, we compute an over-approximation of φ_{reach} by a superset $\varphi^{\#}_{reach}$. Then, we check whether $\varphi^{\#}_{reach}$ contains any error states. If $\varphi^{\#}_{reach} \wedge \varphi_{err} \models false$ holds then $\varphi_{reach} \wedge \varphi_{err} \models false$. Hence the program is safe.

Similarly to the iterative computation of φ_{reach}, we compute $\varphi^{\#}_{reach}$ by applying iteration. However, instead of iteratively applying the post-condition function *post* we use its over-approximation $post^{\#}$ such that

$$\forall \varphi \, \forall \rho : post(\varphi, \rho) \models post^{\#}(\varphi, \rho). \tag{12}$$

We decompose the computation of $post^{\#}$ into two steps. First, we apply *post* and then we over-approximate the result using a function α such that

$$\forall \varphi : \varphi \models \alpha(\varphi). \tag{13}$$

Table 1 Predicate abstraction example

	at_ℓ_1	at_ℓ_2	at_ℓ_3	at_ℓ_4	at_ℓ_5	$y \geq z$	$x \geq y$
$at\_\ell_2 \wedge y \geq z \wedge x + 1 \leq y$	$\not\models$	$\models$	$\not\models$	$\not\models$	$\not\models$	$\models$	$\not\models$

That is, given an over-approximating function α we define $post^{\#}$ as follows.

$$post^{\#}(\varphi, \rho) = \alpha\big(post(\varphi, \rho)\big). \tag{14}$$

Finally, we compute $\varphi^{\#}_{reach}$ as follows.

$$\begin{aligned} \varphi^{\#}_{reach} &= \alpha(\varphi_{init}) \\ &\quad \vee post^{\#}\big(\alpha(\varphi_{init}), \rho_{\mathcal{P}}\big) \\ &\quad \vee post^{\#}\big(post^{\#}\big(\alpha(\varphi_{init}), \rho_{\mathcal{P}}\big), \rho_{\mathcal{P}}\big) \vee \ldots \\ &= \bigvee_{i \geq 0} \big(post^{\#}\big)^i \big(\alpha(\varphi_{init}), \rho_{\mathcal{P}}\big). \end{aligned} \tag{15}$$

We will formalize the over-approximation-based reachability computation through the iterated application of the abstract post-condition operator as indicated by Eq. (15). Its result contains the set of reachable program states. Formally, $\varphi_{reach} \models \varphi^{\#}_{reach}$.

Predicate Abstraction

We construct an over-approximation using a finite number of building blocks, the so-called predicates $p_1, \ldots, p_n$. Each predicate is set of states denoted by a formula over the program variables V.

We fix a finite set of predicates $Preds = \{p_1, \ldots, p_n\}$. Given $Preds$, we can construct an over-approximation of φ as follows [23, 31].

$$\alpha(\varphi) = \bigwedge \{p \in Preds \mid \varphi \models p\}. \tag{16}$$

That is, the over-approximating function α maps a set of states φ to the conjunction of all predicates that are entailed by φ.

If the set of entailed predicates is empty then the result of applying predicate abstraction is $\bigwedge \emptyset$ and is equivalent to *true*.

Example 11 Consider a set of predicates $Preds = \{at\_\ell_1, \ldots, at\_\ell_5, y \geq z, x \geq y\}$. We compute $\alpha(at\_\ell_2 \wedge y \geq z \wedge x + 1 \leq y)$ as follows. First, we check the logical consequence between the argument to the abstraction function and each of the predicates. The results are presented in Table 1.

Then, we take the conjunction of the entailed predicates as the result of the abstraction.

$$\alpha(at\_\ell_2 \wedge y \geq z \wedge x + 1 \leq y) = \bigwedge \{at\_\ell_2, y \geq z\} = at\_\ell_2 \wedge y \geq z. \qquad \square$$

The predicate abstraction function in Eq. (16) approximates φ using a conjunction of predicates, which requires n entailment checks where n is the number of given predicates.

Example 12 We use predicate abstraction to compute $\varphi^{\#}_{reach}$ for our example program following the iterative scheme presented in Eq. (15). Let $Preds = \{false, at\_\ell_1, \ldots, at\_\ell_5, y \geq z, x \geq y\}$. First, let φ_1 be the over-approximation of the set of initial states φ_{init}:

$$\varphi_1 = \alpha(at\_\ell_1) = \bigwedge\{at\_\ell_1\} = at\_\ell_1.$$

We apply $post^{\#}$ on φ_1 w.r.t. each program transition and obtain

$$\varphi_2 = post^{\#}(\varphi_1, \rho_1) = \alpha(\underbrace{at\_\ell_2 \wedge y \geq z}_{post(\varphi_1,\rho_1)}) = \bigwedge\{at\_\ell_2, y \geq z\} = at\_\ell_2 \wedge y \geq z,$$

whereas $post^{\#}(\varphi_1, \rho_2) = \cdots = post^{\#}(\varphi_1, \rho_5) = \bigwedge\{false, \ldots\} = false$.

Now we apply program transitions on φ_2 using $post^{\#}$. The application of ρ_1, ρ_4, and ρ_5 on φ_2 result in *false* for the following reason. φ_2 requires at_ℓ_2, but the transition relations ρ_1, ρ_4, and ρ_5 are applicable if either at_ℓ_1 or at_ℓ_3 holds. For ρ_2 we obtain

$$post^{\#}(\varphi_2, \rho_2) = \alpha(at\_\ell_2 \wedge y \geq z \wedge x \leq y) = \bigwedge\{at\_\ell_2, y \geq z\} = at\_\ell_2 \wedge y \geq z.$$

The resulting set above is equal to φ_2 and, therefore, is discarded, since we are already exploring states reachable from φ_2. For ρ_3 we obtain

$$\begin{aligned} post^{\#}(\varphi_2, \rho_3) &= \alpha(at\_\ell_3 \wedge y \geq z \wedge x \geq y) \\ &= \bigwedge\{at\_\ell_3, y \geq z, x \geq y\} = at\_\ell_3 \wedge y \geq z \wedge x \geq y \\ &= \varphi_3. \end{aligned}$$

We compute an over-approximation of the set of states that are reachable from φ_3 by applying $post^{\#}$. The transitions ρ_1, ρ_2, and ρ_3 result in *false* due to an inconsistency caused by the program counter valuations in φ_3 and the respective transition relations. For the transition ρ_4 we obtain

$$\begin{aligned} post^{\#}(\varphi_3, \rho_4) &= \alpha(at\_\ell_4 \wedge y \geq z \wedge x \geq y \wedge x \geq z) \\ &= \bigwedge\{at\_\ell_4, y \geq z, x \geq y\} = at\_\ell_4 \wedge y \geq z \wedge x \geq y \\ &= \varphi_4. \end{aligned}$$

For the transition ρ_5, which corresponds to the assertion violation, we obtain

$$\begin{aligned} post^{\#}(\varphi_3, \rho_5) &= \alpha(at\_\ell_5 \wedge y \geq z \wedge x \geq y \wedge x + 1 \leq z) \\ &= false. \end{aligned}$$

Fig. 2 Algorithm ABSTREACH for abstract reachability computation w.r.t. a given finite set of predicates

```
    function AbstReach
    input
      Preds - predicates
    begin
1     α := λφ . ⋀{p ∈ Preds | φ ⊨ p}
2     post# := λ(φ, ρ) . α(post(φ, ρ))
3     ReachStates# := {α(φ_init)}
4     Parent := ∅
5     Worklist := ReachStates#
6     while Worklist ≠ ∅ do
7       φ := choose from Worklist
8       Worklist := Worklist \ {φ}
9       for each ρ ∈ T do
10        φ' := post#(φ, ρ)
11        if φ' ⊭ ⋁ ReachStates# then
12          ReachStates# := {φ'} ∪ ReachStates#
13          Parent := {(φ, ρ, φ')} ∪ Parent
14          Worklist := {φ'} ∪ Worklist
15    return (ReachStates#, Parent)
    end
```

Any further application of program transitions does not compute any additional reachable states. We conclude that $\varphi^{\#}_{reach} = \varphi_1 \vee \cdots \vee \varphi_4$. Furthermore, since $\varphi^{\#}_{reach} \wedge at\_\ell_5 \models false$ the program is safe. □

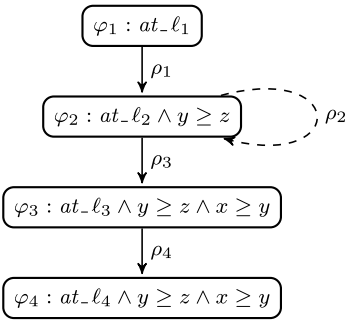
Algorithm ABSTREACH

We combine the characterization of abstract reachability using Eq. (15) with the predicate abstraction function given in Eq. (16) and obtain an algorithm ABSTREACH for computing *ReachStates*#. The algorithm is shown in Fig. 2.

ABSTREACH takes as input a finite set of predicates *Preds* and computes a set of formulas *ReachStates*# that represents an over-approximation $\varphi^{\#}_{reach}$. Furthermore, ABSTREACH records its intermediate computation steps in a labeled tree *Parent*. (In the next section we will show how this tree can be used to discover new predicates when a refined abstraction is needed.)

The initialization steps of ABSTREACH are shown in lines 1–5 of Fig. 2. First, we construct the abstraction function α according to Eq. (16), and then use it to construct an over-approximation *post*# of the post-condition function according to Eq. (14). We initialize *ReachStates*# with an over-approximation of the initial program states, which corresponds to the first disjunct in Eq. (15). Since the initial states do not have any predecessors, *Parent* is initially empty. Finally, we create a worklist *Worklist* that contains sets of states on which *post*# has not been applied yet.

The main part of ABSTREACH in lines 6–14 implements the iterative application of *post*# in Eq. (15) using a while loop. The loop termination condition checks whether *Worklist* has any items to process. In case the worklist is not empty, we



$$\varphi_1 = \alpha(\varphi_{init})$$
$$\varphi_2 = post^{\#}(\varphi_1, \rho_1)$$
$$post^{\#}(\varphi_2, \rho_2) \models \varphi_2$$
$$\varphi_3 = post^{\#}(\varphi_2, \rho_3)$$
$$\varphi_4 = post^{\#}(\varphi_3, \rho_4)$$

Fig. 3 Applying ABSTREACH on the program in Fig. 1 and the set of predicates $Preds = \{false, at\_\ell_1, \ldots, at\_\ell_5, y \geq z, x \geq y\}$. The nodes $\varphi_1, \ldots, \varphi_4$ represent elements of $ReachStates^{\#}$. *Labeled edges* connecting the nodes represent *Parent*. The *dotted edge* denotes the entailment relation between $post^{\#}(\varphi_2, \rho_2)$ and φ_2

choose such an item, say φ, and remove it from the worklist. For brevity, we leave the selection procedure unspecified, but note that various strategies are possible, e.g., breadth- or depth-first search. Then, we apply $post^{\#}$ w.r.t. each program transition, say ρ, on φ. Let φ' be the result of such an application. We add φ' to $ReachStates^{\#}$ if φ' contains some program states that are not already contained in one of the formulas in $ReachStates^{\#}$. We formulate the above test as an entailment check between φ' and the disjunction of all formulas in $ReachStates^{\#}$. Often, there is a formula ψ in $ReachStates^{\#}$ such that $\varphi' \models \psi$. Otherwise, that φ is added to $ReachStates^{\#}$, and we record that φ' was computed by applying ρ on φ by adding a tuple $(\varphi, \rho, \varphi')$ to *Parent*. Finally, φ' is put on the worklist.

The loop execution terminates after a finite number of steps, since the range of $post^{\#}$ is finite (and is of size 2^n where n is the size of *Preds*). The disjunction of formulas in $ReachStates^{\#}$ is logically equivalent to $\varphi^{\#}_{reach}$.

Example 13 We describe the application of ABSTREACH on our example program when $Preds = \{false, at\_\ell_1, \ldots, at\_\ell_5, y \geq z, x \geq y\}$. Figure 3 provides a pictorial illustration. Example 12 provides details on computed over-approximations of post-conditions.

After constructing α and $post^{\#}$ for the given predicates, we compute $\varphi_1 = (at\_\ell_1)$ and put it into $ReachStates^{\#}$ and into *Worklist*. See the node φ_1 in Fig. 3.

During the first loop iteration, we choose φ_1 to be the element taken from the worklist. Now we compute $post^{\#}$ w.r.t. each program transition. For ρ_1 we obtain $\varphi_2 = (at\_\ell_2 \wedge y \geq z)$. The entailment check $\varphi_2 \models \bigvee ReachStates^{\#}$ fails, since $\bigvee ReachStates^{\#}$ is equal to φ_1 and $\varphi_2 \not\models \varphi_1$. Hence, φ_2 is added to $ReachStates^{\#}$. As a result, the tuple $(\varphi_1, \rho_1, \varphi_2)$ is added to *Parent* and φ_2 becomes a worklist item. See the node φ_2 as well as the edge between φ_1 and φ_2 in Fig. 3. We continue with applying program transitions on φ_1. For ρ_2 we obtain $post^{\#}(\varphi_1, \rho_2) = false$. Since $false \models \bigvee ReachStates^{\#}$ there is no addition to $ReachStates^{\#}$. Similarly, applying $\rho_3, \ldots, \rho_5$ does not modify $ReachStates^{\#}$.

We start the second loop iteration with $ReachStates^{\#} = \{\varphi_1, \varphi_2\}$, $Worklist = \{\varphi_2\}$, and $Parent = \{(\varphi_1, \rho_1, \varphi_2)\}$. We choose φ_2 from the worklist. When applying $post^{\#}$ on φ_2 only ρ_2 and ρ_3 result in sets of successor states that are not equal to *false*. We obtain $post^{\#}(\varphi_2, \rho_2) = (at\_\ell_2 \wedge y \geq z)$. Since $(at\_\ell_2 \wedge y \geq z)$ entails φ_2 and hence $\bigvee ReachStates^{\#}$, nothing is added to $ReachStates^{\#}$ and we proceed directly with ρ_3. For $\varphi_3 = post^{\#}(\varphi_2, \rho_3) = (at\_\ell_3 \wedge y \geq z \wedge x \geq y)$ we observe that $\varphi_3 \not\models \bigvee ReachStates^{\#}$. Hence, we add φ_3 to $ReachStates^{\#}$ and $Worklist$, while $(\varphi_2, \rho_3, \varphi_3)$ is recorded in $Parent$. See the node φ_3 as well as the edge between φ_2 and φ_3 in Fig. 3.

At the beginning of the third loop iteration we have $ReachStates^{\#} = \{\varphi_1, \varphi_2, \varphi_3\}$, $Worklist = \{\varphi_3\}$, and $Parent = \{(\varphi_1, \rho_1, \varphi_2), (\varphi_2, \rho_3, \varphi_3)\}$. We choose φ_3 from the worklist. After computing φ_4 by applying ρ_4 and discovering that $\varphi_4 \not\models \bigvee ReachStates^{\#}$, we add φ_4 following the algorithm. See the node φ_4 as well as the edge between φ_3 and φ_4 in Fig. 3. Since all other program transitions yield *false* we proceed with the next iteration.

The fourth loop iteration removes φ_4 from the worklist, but does not add any new elements to it. Hence AbstReach terminates and outputs $ReachStates^{\#} = \{\varphi_1, \ldots, \varphi_4\}$ as well as $Parent = \{(\varphi_1, \rho_1, \varphi_2), (\varphi_2, \rho_3, \varphi_3), (\varphi_3, \rho_4, \varphi_4)\}$. □

Discussion

We have presented predicate abstraction and the abstract reachability algorithm in the framework of *abstract interpretation* [23] (where predicate abstraction is formalized through the concept of *Moore families*). This presentation is very general. It allows us to leave open many design choices.

One such design choice, conceptually and technically, is to split the algorithm into two steps. The first ("offline") step is to compute the function $post^{\#}$. The second step is to iterate $post^{\#}$ until a fixpoint is reached. Often, e.g., in [2–4, 17], conjunctions of predicates are viewed as *abstract states* (which can possibly be represented as bitvectors). Instead of constructing the function $post^{\#}$ directly, one may first construct a relation between abstract states. If one views this relation as the transition relation of an *abstract program* $\mathcal{P}^{\#}$ (the "abstraction of the program $\mathcal{P}$"), then the abstraction of the post operator for the program $\mathcal{P}$ can be phrased as the post operator of the (finite-state) abstract program $\mathcal{P}^{\#}$, formally

$$post^{\#}_{\mathcal{P}} = post_{\mathcal{P}^{\#}}.$$

Our definition of the over-approximating function α in Eq. (16) combines predicates into a conjunction. That is, the over-approximating function α maps a set of states φ to the strongest conjunction e of predicates p in $Preds$ such that e is still entailed by φ. The name *Cartesian abstraction* was coined for this approach in [3] (in analogy to the abstraction of a set of tuples by the Cartesian product of its component sets; each predicate is treated independently from the others, like the components in the tuples in the Cartesian product). The alternative approach of *Boolean abstraction* is to combine predicates into a Boolean expression (of a general form). Then, the over-approximating function α maps a set of states φ to the

strongest Boolean expression e over the given set of predicates such that that e is still entailed by φ. The gain in precision is usually not worth the higher cost of computing the abstraction function. We point out, however, the technique of so-called *large-block encoding* which operates on compound program transitions with rich Boolean structure and which can leverage the advances in state-of-the-art decision procedures and thus can offer both precision and efficiency [8, 11].

15.5.2 Termination and Transition Predicate Abstraction

In this section, we show how predicate abstraction can be used for computing transition invariants, and thus proving program termination.

In principle, transition invariants can be computed by applying the iterative scheme in Eq. (10) and then restricting the obtained result to reachable states by relying on Eq. (11). The iteration of *comp* finishes when no new pair of program states is discovered. Unfortunately, such an iteration process does not terminate in finite time, for similar reasons to those presented in Sect. 15.5.1.

Instead of computing ψ_{ti} we compute its over-approximation by a superset $\psi_{ti}^{\#}$. Then, we check whether $\psi_{ti}^{\#}$ is disjunctively well-founded. If $\psi_{ti}^{\#}$ satisfies the disjunctive well-foundedness condition then ψ_{ti} is disjunctively well-founded as well. Hence the program terminates.

Similarly to the computation of ψ_{ti}, we compute $\psi_{ti}^{\#}$ by applying iteration. However, instead of iteratively applying the relational composition function *comp* we use its over-approximation $comp^{\#}$ such that

$$\forall \psi : comp(\psi) \models comp^{\#}(\psi). \tag{17}$$

We decompose the computation of $comp^{\#}$ into two steps. First, we apply *comp* and then we over-approximate the result using a function $\ddot{\alpha}$ such that

$$\forall \psi : \psi \models \ddot{\alpha}(\psi). \tag{18}$$

That is, given an over-approximating function $\ddot{\alpha}$ we define $comp^{\#}$ as follows.

$$comp^{\#}(\psi) = \ddot{\alpha}\big(comp(\psi)\big). \tag{19}$$

Finally, we can obtain $\psi_{ti}^{\#}$ by using a previously computed over-approximation of reachable states $\varphi_{reach}^{\#}$ as follows.

$$\begin{aligned} \psi_{ti}^{\#} &= comp^{\#}\big(\varphi_{reach}^{\#} \wedge V' = V\big) \\ &\quad \vee comp^{\#}\big(comp^{\#}\big(\varphi_{reach}^{\#} \wedge V' = V\big)\big) \vee \ldots \\ &= \bigvee_{i \geq 1} \big(comp^{\#}\big)^{i}\big(\varphi_{reach}^{\#} \wedge V' = V\big). \end{aligned} \tag{20}$$

We formalize our over-approximation-based transition invariant computation as follows. The strongest transition invariant of the program is contained in the result of the abstract computation given by Eq. (20). Formally, $\psi_{ti} \models \psi_{ti}^{\#}$.

Transition Predicate Abstraction

We construct an over-approximation $\psi_{ti}^{\#}$ using a given set of building blocks, so-called *transition predicates*. Each transition predicate is a formula over the program variables V and their primed verions V', which represents a binary relation over program states [53].

We fix a finite set of *transition predicates* $\mathit{TransPreds} = \{\ddot{p}_1, \ldots, \ddot{p}_n\}$. Then, we define an over-approximation of ψ that is constructed using *TransPreds* as follows.

$$\ddot{\alpha}(\psi) = \bigwedge \{\ddot{p} \in \mathit{TransPreds} \mid \psi \models \ddot{p}\}. \tag{21}$$

If the set of entailed transition predicates is empty then the result of applying transition predicate abstraction is $\bigwedge \emptyset$, which is equivalent to *true*.

Example 14 Consider a set of predicates $\mathit{TransPreds} = \{at\_\ell_1, \ldots, at\_\ell_5, at'\_\ell_1, \ldots, at'\_\ell_5, x \geq 0, x' \geq x + 1, x' \geq x, y' \geq y, y' \geq y + 1\}$. We will apply transition predicate abstraction on the transition ρ_2 in the program shown in Fig. 1. We compute $\ddot{\alpha}(\mathit{goto}(\ell_2, \ell_2) \wedge x + 1 \leq y \wedge x' = x + 1 \wedge \mathit{unchanged}(y, z))$ as follows. First, we check the logical consequence between the argument to the abstraction function and each of the predicates and obtain the following set of entailed predicates:

$$\{at\_\ell_2, at'\_\ell_2, x \geq 0, x' \geq x + 1, x' \geq x, y' \geq y\}.$$

Then, we take the conjunction of the entailed predicates as the result of the abstraction.

$$\begin{aligned} &\bigwedge \{at\_\ell_2, at'\_\ell_2, x \geq 0, x' \geq x + 1, x' \geq x, y' \geq y\} \\ &= at\_\ell_2 \wedge at'\_\ell_2 \wedge x \geq 0 \wedge x' \geq x + 1 \wedge y' \geq y. \end{aligned}$$

□

The transition predicate abstraction function in Eq. (21) approximates ψ using a conjunction of transition predicates, which requires n entailment checks where n is the number of given transition predicates.

Example 15 We use transition predicate abstraction to compute $\psi_{ti}^{\#}$ for our example program from Fig. 1 following the iterative scheme presented in Eq. (20). Since we are interested in proving termination we will ignore the assertion statement occurring in the program. As a consequence, in this example we will not take transitions ρ_4 and ρ_5 into account.

For simplicity, we use $\varphi_{\mathit{reach}}^{\#} = \mathit{true}$, which states that the set of reachable states is contained in the set of all possible program states represented by *true*. Let $\mathit{TransPreds} = \{\mathit{false}, at\_\ell_1, \ldots, at\_\ell_3, at'\_\ell_1, \ldots, at'\_\ell_3, x \leq y, y' - x' \leq y - x - 1\}$.

First, we compute the first step of the over-approximation $comp^{\#}(\varphi^{\#}_{reach} \wedge V' = V)$:

$$\begin{aligned} comp^{\#}\big(\varphi^{\#}_{reach} \wedge V' = V\big) &= \ddot{\alpha}\big(\big(\varphi^{\#}_{reach} \wedge V' = V\big) \circ \rho_{\mathcal{P}}\big) \\ &= \ddot{\alpha}\big(\big(\varphi^{\#}_{reach} \wedge V' = V\big) \circ (\rho_1 \vee \rho_2 \vee \rho_3)\big) \\ &= \ddot{\alpha}(\rho_1 \vee \rho_2 \vee \rho_3). \end{aligned}$$

We obtain the following abstractions for each of the transitions:

$$\begin{aligned} \psi_1 &= \ddot{\alpha}(\rho_1) = goto(\ell_1, \ell_2), \\ \psi_2 &= \ddot{\alpha}(\rho_2) = \big(goto(\ell_2, \ell_2) \wedge x \leq y \wedge y' - x' \leq y - x - 1\big), \\ \psi_3 &= \ddot{\alpha}(\rho_3) = goto(\ell_2, \ell_3). \end{aligned}$$

We apply $comp^{\#}$ on $\psi_1, \ldots, \psi_3$ and obtain the following non-empty abstractions.

$$\begin{aligned} \psi_4 &= \ddot{\alpha}(\psi_1 \circ \rho_2) = goto(\ell_1, \ell_2), \\ \psi_5 &= \ddot{\alpha}(\psi_1 \circ \rho_3) = goto(\ell_1, \ell_3), \\ \psi_6 &= \ddot{\alpha}(\psi_2 \circ \rho_2) = \big(goto(\ell_2, \ell_2) \wedge x \leq y \wedge y' - x' \leq y - x - 1\big), \\ \psi_7 &= \ddot{\alpha}(\psi_2 \circ \rho_3) = \big(goto(\ell_2, \ell_3) \wedge x \leq y \wedge y' - x' \leq y - x - 1\big). \end{aligned}$$

We observe that $\psi_4 \models \psi_1$, $\psi_6 \models \psi_2$, and $\psi_7 \models \psi_3$, hence, ψ_4, ψ_6, and ψ_7 can be ignored. We apply $comp^{\#}$ on ψ_5 and observe that the resulting abstractions are empty. Hence, we finish the iterative computation and obtain

$$\psi^{\#}_{ti} = \psi_1 \vee \psi_2 \vee \psi_3 \vee \psi_5.$$

The computed transition invariant $\psi^{\#}_{ti}$ is disjunctively well-founded. Each disjunct in $\psi^{\#}_{ti}$ whose start and finish locations are not equal, e.g., ψ_1 with the start location ℓ_1 and finish location ℓ_2, is well-founded. The remaining disjunct ψ_2 is well-founded since the value of $y - x$ is greater than zero whenever ψ_2 makes a step and decreases during each step of ψ_2. Hence, we conclude that $\psi^{\#}_{ti}$ is contained in a finite union of well-founded relations. Thus, the program terminates.

We represent the above computation pictorially in Fig. 4. □

Algorithm TRANSABSTREACH

We combine the characterization of abstract transition invariants using Eq. (20) with the transition predicate abstraction function given in Eq. (21) and obtain an algorithm TRANSABSTREACH for computing $ReachTrans^{\#}$. The algorithm is shown in Fig. 5.

TRANSABSTREACH takes as input a finite set of transition predicates *TransPreds* and computes a set of formulas $ReachTrans^{\#}$ that represents an over-approximation $\psi^{\#}_{ti}$. Furthermore, TRANSABSTREACH records its intermediate computation steps in a labeled tree *TransParent*. In the next section we will show how this tree

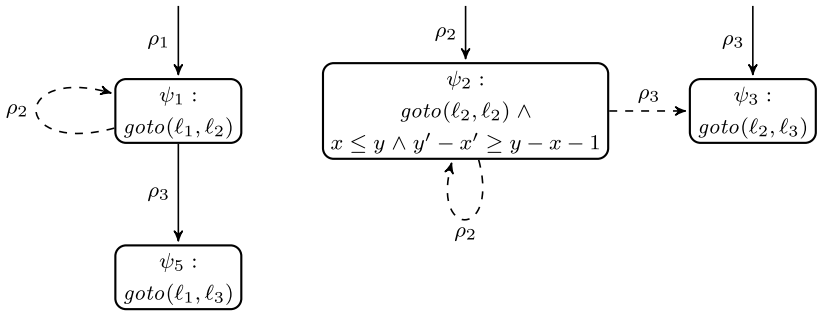


Fig. 4 Abstract transitions computed in Example 15. *Solid edges* connecting the nodes represent how nodes were computed. *Dotted edges* denote the entailment relation

```
    function TransAbstReach
    input
      ReachStates# - reachable abstract states
      TransPreds - transition predicates
    begin
1     α̈ := λφ̈ . ⋀{p̈ ∈ TransPreds | φ̈ ⊨ p̈}
2     ReachTrans# := ∅
3     TransParent := ∅
4     Worklist := {φ ∧ unchanged(V) | φ ∈ ReachStates#}
5     while Worklist ≠ ∅ do
6       φ̈ := choose from Worklist
7       Worklist := Worklist \ {φ̈}
8       for each ρ ∈ 𝒯 do
9         φ̈′ := α̈(φ̈ ∘ ρ)
10        if ¬(∃ψ̈ ∈ ReachTrans# : φ̈′ ⊨ ψ̈) then
11          ReachTrans# := {φ̈} ∪ ReachTrans#
12          TransParent := {(φ̈, ρ, φ̈′)} ∪ TransParent
13          Worklist := {φ̈′} ∪ Worklist
14    return (ReachTrans#, TransParent)
    end
```

Fig. 5 Abstract transitive closure computation

can be used to discover new transition predicates when a refined abstraction is needed.

The initialization steps of TransAbstReach are shown in lines 1–4 in Fig. 5. First, we construct the abstraction function $\ddot{\alpha}$ according to Eq. (21), and then use it to construct an over-approximation of the relational composition of program transitions. *ReachTrans*$^\#$ is initially empty, which corresponds to the fact that we are interested in the irreflexive transitive closure following Eq. (20). Since the initial relations do not have any predecessors, *TransParent* is initially empty. Finally, we

initialize the worklist *Worklist* with a set of identity relations over program states that are restricted to the sets of states represented by elements of $ReachStates^{\#}$. Such initialization together with the first iteration of the while loop correspond to the first disjunct in Eq. (20).

The main part of AbstReach in lines 5–13 implements the iterative application of $comp^{\#}$ in Eq. (20) using a while loop. Since we are interested in applying individual program transitions one by one, we rely on a direct application of relational composition $\circ$ and transition predicate abstraction $\ddot{\alpha}$. The loop termination condition checks whether *Worklist* has any items to process. In case the worklist is not empty, we choose such an item, say $\ddot{\varphi}$, and remove it from the worklist. For brevity, we leave the selection procedure unspecified, but note that various strategies are possible, e.g., breadth- or depth-first search. Then, we apply $\circ$ and $\ddot{\alpha}$ w.r.t. each program transition, say ρ, on $\ddot{\varphi}$. Let $\ddot{\varphi}'$ be the result of such an application. We add $\ddot{\varphi}'$ to $ReachTrans^{\#}$ if $\ddot{\varphi}'$ contains some pairs of program states that are not already contained in one of the formulas in $ReachTrans^{\#}$. We formulate the above test as an entailment check between $\ddot{\varphi}'$ and the disjunction of all formulas in $ReachTrans^{\#}$. Often, there is a formula $\ddot{\psi}$ in $ReachStates^{\#}$ such that $\ddot{\varphi}' \models \ddot{\psi}$. Otherwise, $\ddot{\varphi}$ is added to $ReachTrans^{\#}$, and we record that $\ddot{\varphi}'$ was computed by applying ρ on $\ddot{\varphi}$ by adding a tuple $(\ddot{\varphi}, \rho, \ddot{\varphi}')$ to *TransParent*. Finally, $\ddot{\varphi}'$ is put on the worklist.

The loop execution terminates after a finite number of steps, since the range of $\ddot{\alpha}$ is finite (and is of size 2^n where n is the size of *TransPreds*). The disjunction of formulas in $ReachTrans^{\#}$ is logically equivalent to $\psi_{ti}^{\#}$.

Example 16 We illustrate TransAbstReach by showing how it automates the computation presented in Example 15. We again consider our example program from Fig. 1 together with a set of transition predicates $TransPreds = \{false, at\_\ell_1, \ldots, at\_\ell_3, at'\_\ell_1, \ldots, at'\_\ell_3\}$ and an over-approximation of reachable states $ReachStates^{\#} = \{true\}$. After executing the initialization steps in TransAbstReach we obtain $ReachTrans^{\#} = \emptyset$, $TransParent = \emptyset$, and $Worklist = \{true \wedge unchanged(x, y, z)\}$.

The first iteration of the while loop chooses $\ddot{\varphi} = (true \wedge unchanged(x, y, z))$ and removes it from *Worklist*. Now TransAbstReach iterates through the transitions of the program. First, we consider $\rho = \rho_1$ and obtain

$$\ddot{\varphi}' = \ddot{\alpha}\big(\big(true \wedge unchanged(x, y, z)\big) \circ \rho_1\big) = goto(\ell_1, \ell_2).$$

Since $ReachTrans^{\#} = \emptyset$, we obtain

$$\begin{aligned} ReachTrans^{\#} &= \big\{goto(\ell_1, \ell_2)\big\}, \\ TransParent &= \big\{\big(true \wedge unchanged(x, y, z), \rho_1, goto(\ell_1, \ell_2)\big)\big\}, \\ Worklist &= \big\{goto(\ell_1, \ell_2)\big\}, \end{aligned}$$

and proceed with the remaining program transitions. For $\rho = \rho_2$ we obtain $\ddot{\varphi}' = goto(\ell_2, \ell_2)$. Since $\ddot{\varphi}'$ does not entail the transition relation already contained in

ReachTrans$^{\#}$, we obtain (in this example, "…" denotes the previously assigned value)

$$\begin{aligned} \mathit{ReachTrans}^{\#} &= \{\mathit{goto}(\ell_2, \ell_2), \dots\}, \\ \mathit{TransParent} &= \{(\mathit{true} \wedge \mathit{unchanged}(x, y, z), \rho_2, \mathit{goto}(\ell_2, \ell_2), \dots\}, \\ \mathit{Worklist} &= \{\mathit{goto}(\ell_2, \ell_2), \mathit{goto}(\ell_1, \ell_2)\}. \end{aligned}$$

After applying $\rho = \rho_3$ we obtain $\ddot{\varphi}' = \mathit{goto}(\ell_2, \ell_3)$, which leads to

$$\begin{aligned} \mathit{ReachTrans}^{\#} &= \{\mathit{goto}(\ell_2, \ell_3), \dots\}, \\ \mathit{TransParent} &= \{(\mathit{true} \wedge \mathit{unchanged}(x, y, z), \rho_3, \mathit{goto}(\ell_2, \ell_3)), \dots\}, \\ \mathit{Worklist} &= \{\mathit{goto}(\ell_2, \ell_3), \mathit{goto}(\ell_2, \ell_2), \mathit{goto}(\ell_1, \ell_2)\}. \end{aligned}$$

Now, we proceed with the second iteration of the while loop. We choose $\ddot{\varphi} = \mathit{goto}(\ell_1, \ell_2)$ and proceed with applying program transitions. Applying ρ_1 yields $\ddot{\varphi}' = \mathit{false}$. For $\rho = \rho_2$ we obtain $\ddot{\varphi} = \mathit{goto}(\ell_1, \ell_2)$. Since there exists an element of *ReachTrans*$^{\#}$ that is entailed by $\ddot{\varphi}'$, namely $\mathit{goto}(\ell_1, \ell_2)$, the computed $\ddot{\varphi}'$ is discarded. Applying ρ_3 yields $\ddot{\varphi}' = \mathit{goto}(\ell_1, \ell_3)$ and leads to

$$\begin{aligned} \mathit{ReachTrans}^{\#} &= \{\mathit{goto}(\ell_1, \ell_3), \dots\}, \\ \mathit{TransParent} &= \{(\mathit{goto}(\ell_1, \ell_2), \rho_3, \mathit{goto}(\ell_1, \ell_3)), \dots\}, \\ \mathit{Worklist} &= \{\mathit{goto}(\ell_1, \ell_3), \mathit{goto}(\ell_2, \ell_3), \mathit{goto}(\ell_2, \ell_2)\}. \end{aligned}$$

Subsequent iterations of the while loop proceed similarly and modify neither *ReachTrans*$^{\#}$ nor *TransParent*. Finally, *Worklist* becomes empty and TRANSABSTREACH terminates. We obtain the following output:

$$\begin{aligned} \mathit{ReachTrans}^{\#} = \{&\mathit{goto}(\ell_1, \ell_3), \mathit{goto}(\ell_2, \ell_3), \mathit{goto}(\ell_2, \ell_2), \mathit{goto}(\ell_1, \ell_2)\}, \\ \mathit{TransParent} = \{&(\mathit{goto}(\ell_1, \ell_2), \rho_3, \mathit{goto}(\ell_1, \ell_3)), \\ &(\mathit{true} \wedge \mathit{unchanged}(x, y, z), \rho_3, \mathit{goto}(\ell_2, \ell_3)), \\ &(\mathit{true} \wedge \mathit{unchanged}(x, y, z), \rho_2, \mathit{goto}(\ell_2, \ell_2)), \\ &(\mathit{true} \wedge \mathit{unchanged}(x, y, z), \rho_1, \mathit{goto}(\ell_1, \ell_2))\}. \end{aligned}$$

□

15.6 Abstraction Refinement

The algorithm ABSTREACH requires a set of predicates in order to compute an over-approximation of the set of reachable program states. Similarly, the algorithm TRANSABSTREACH requires a set of transition predicates in order to compute an

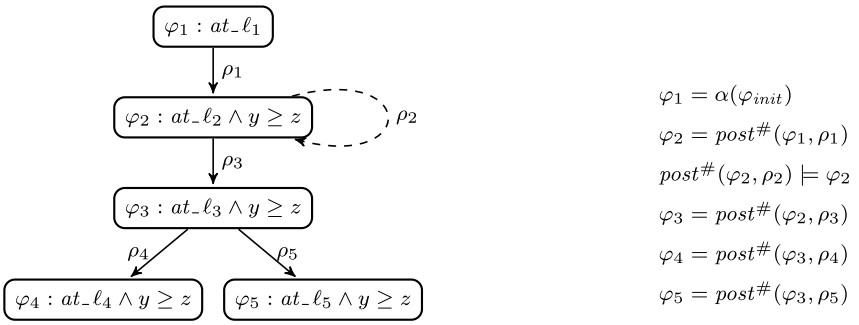


Fig. 6 Abstract reachability computation with $Preds = \{false, at\_\ell_1, \ldots, at\_\ell_5, y \geq z\}$

over-approximation of the set of reachable pairs of program states. Finding the right set of predicates (or of transition predicates) that yields a sufficiently precise over-approximation is a difficult task.

15.6.1 Refinement of Predicate Abstraction

The procedure for refining predicate abstraction considers certain program paths as the main source of information. By exploring such paths we can obtain an adequate set of predicates to prove the program correct.

Analysis of Counterexample Paths

We start with an example that illustrates the impact of over-approximation and how it can be eliminated.

Example 17 In Example 13, the provided set of predicates was adequate for proving program safety. Omitting just one predicate, e.g., provide the predicates $Preds = \{false, at\_\ell_1, \ldots, at\_\ell_5, y \geq z\}$ without $x \geq y$, leads to an over-approximation $\varphi^{\#}_{reach}$ that has a non-empty intersection with the error states. As shown in Fig. 6, we have $\varphi_5 \wedge \varphi_{err} \not\models false$. That is, ABSTREACH fails to prove the property without the predicate $x \geq y$.

We analyse the reason for the excessive over-approximation. Figure 6 shows that the *Parent* relation records a sequence of three steps leading to the computation of φ_5. First, we apply ρ_1 to φ_1 and compute φ_2. Then, φ_3 is obtained by applying ρ_3 to φ_2. Finally, ρ_5 is applied to φ_3 and results in φ_5. Thus, we note that the sequence of program transitions ρ_1, ρ_3, and ρ_5 determined φ_5. We refer to this sequence as a counterexample path. Using this path and the functions α and $post^{\#}$ corresponding to the current set of predicates we obtain

$$\varphi_5 = post^{\#}\big(post^{\#}\big(post^{\#}\big(\alpha(\varphi_{init}), \rho_1\big), \rho_3\big), \rho_5\big).$$

Table 2 Example solution to the over-approximation condition

ψ_1	ψ_2	ψ_3	ψ_4
at_ℓ_1	$at\_\ell_2 \wedge y \geq z$	$at\_\ell_3 \wedge x \geq z$	$false$

That is, φ_5 is equal to the over-approximation of the post-condition computed along the counterexample path.

Now we check whether the counterexample path also leads to an error state when no over-approximation is applied. First we compute

$$\begin{aligned} post\big(post\big(post(\varphi_{init}, \rho_1), \rho_3\big), \rho_5\big) &= post\big(post(at\_\ell_2 \wedge y \geq z, \rho_3), \rho_5\big) \\ &= post(at\_\ell_3 \wedge y \geq z \wedge x \geq y, \rho_5) \\ &= false. \end{aligned}$$

Hence, by executing the program transitions ρ_1, ρ_3, and ρ_5 it is not possible to reach any error state. We conclude that the over-approximation is too coarse, at least when dealing with the above path.

We need a more precise over-approximation that will prevent $post^{\#}$ from including states that lead to error states along the path ρ_1, ρ_3, and ρ_5. Concretely, we need a refined abstraction function α and a corresponding $post^{\#}$ such that the execution of AbstReach along the counterexample path does not compute a set of states that contains some error states:

$$post^{\#}\big(post^{\#}\big(post^{\#}\big(\alpha(\varphi_{init}), \rho_1\big), \rho_3\big), \rho_5\big) \wedge \varphi_{err} \models false.$$

We consider the intermediate steps of the above condition and define sets of states $\psi_1, \ldots, \psi_4$ that provide an adequate over-approximation along the path as follows.

$$\begin{aligned} \varphi_{init} &\models \psi_1, \\ post(\psi_1, \rho_1) &\models \psi_2, \\ post(\psi_2, \rho_3) &\models \psi_3, \\ post(\psi_3, \rho_5) &\models \psi_4, \\ \psi_4 \wedge \varphi_{err} &\models false. \end{aligned}$$

The over-approximation given by $\psi_1, \ldots, \psi_4$ is adequate since it guarantees that no error state is reached, while still allowing additional states to be reachable. For example, consider the solution to the above condition given in Table 2.

We can use the obtained solution to refine α in the following way. By adding $\psi_1, \ldots, \psi_4$ to the set of predicates *Preds* we guarantee that the resulting α and $post^{\#}$ are sufficiently precise to show that no error state is reachable along the path ρ_1, ρ_3,

Fig. 7 Path computation

```
function MakePath
input
  ψ - reachable abstract state
  Parent - predecessor relation
begin
1   path := empty sequence
2   φ' := ψ
3   while exist φ and ρ such that (φ, ρ, φ') ∈ Parent do
4     path := ρ . path
5     φ' := φ
6   return path
end
```

and ρ_5. Formally, we obtain

$$\begin{aligned} \alpha(\varphi_{init}) &\models \psi_1, \\ post^{\#}(\psi_1, \rho_1) &\models \psi_2, \\ post^{\#}(\psi_2, \rho_3) &\models \psi_3, \\ post^{\#}(\psi_3, \rho_5) &\models \psi_4, \\ \psi_4 \wedge \varphi_{err} &\models false. \end{aligned}$$

□

We put the above approach for analyzing counterexamples computed by AbstReach into algorithms MakePath, FeasiblePath, and RefinePath.

The algorithm MakePath is shown in Fig. 7. It takes as input a reachable abstract state ψ together with a *Parent* relation. We view *Parent* as a tree where ψ occurs as a node. MakePath outputs a sequence of program transitions that labels the tree edges connecting ψ with the root of the tree. The sequence is constructed iteratively by a backward traversal starting from the input node. The variable *path* keeps track of the construction.

Example 18 For our example tree in Fig. 6 we construct the path by making a call MakePath(φ_5, *Parent*). Then, *path* is extended with the transitions ρ_5, ρ_3, and ρ_1 by considering the edges $(\varphi_3, \rho_5, \varphi_5)$, $(\varphi_2, \rho_3, \varphi_3)$, and $(\varphi_1, \rho_1, \varphi_2)$, respectively. Finally, $path = \rho_1\rho_3\rho_5$ is returned as output. □

The algorithm FeasiblePath is shown in Fig. 8. It takes as input a sequence of program transitions $\rho_1 \ldots \rho_n$ and checks whether there is a computation that is produced by this sequence. The check uses the post-condition function and the relational composition of transitions.

Example 19 When applying FeasiblePath on our example path $\rho_1\rho_3\rho_5$ we obtain the following intermediate results. First, the relational composition of transi-

Fig. 8 Feasibility of a path

```
function FeasiblePath
input
  ρ1 ... ρn - path
begin
1   φ := post(φinit, ρ1 ∘ ... ∘ ρn)
2   if φ ∧ φerr ⊭ false then
3     return true
4   else
5     return false
end
```

tions yields

$$\rho_1 \circ \rho_3 \circ \rho_5 = \mathit{false}.$$

Hence, FeasiblePath sets φ to *false* and then returns *false*. □

The algorithm RefinePath is shown in Fig. 9. It takes as input a sequence of program transitions $\rho_1 \ldots \rho_n$ and computes sets of states $\varphi_0, \ldots, \varphi_n$ satisfying the following conditions. First, we have $\varphi_{init} \models \varphi_0$ and $\varphi_n \wedge \varphi_{err} \models \mathit{false}$. Then, for each $i \in 1..n$ we obtain $\mathit{post}(\varphi_{i-1}, \rho_i) \models \varphi_i$. Thus, $\varphi_0, \ldots, \varphi_n$ computed by RefinePath can be used for refining predicate abstraction. If $\varphi_0, \ldots, \varphi_n$ are added to *Preds* then the resulting α and $\mathit{post}^{\#}$ guarantee that the following conditions hold.

$$\begin{aligned} \alpha(\varphi_{init}) &\models \varphi_0, \\ \mathit{post}^{\#}(\varphi_0, \rho_1) &\models \varphi_1, \\ &\ldots, \\ \mathit{post}^{\#}(\varphi_{n-1}, \rho_n) &\models \varphi_n, \\ \varphi_n \wedge \varphi_{err} &\models \mathit{false}. \end{aligned}$$

```
function RefinePath
input
  ρ1 ... ρn - path
begin
1   φ0, ..., φn := compute such that
2     (φinit ⊨ φ0) ∧
3     (post(φ0, ρ1) ⊨ φ1) ∧ ... ∧ (post(φn−1, ρn) ⊨ φn) ∧
4     (φn ∧ φerr ⊨ false)
5   return {φ0, ..., φn}
end
```

Fig. 9 Counterexample-guided discovery of predicates

```
    function AbstRefineLoop
    begin
1     Preds := ∅
2     repeat
3       (ReachStates#, Parent) := AbstReach(Preds)
4       if exists ψ ∈ ReachStates# such that ψ ∧ φerr ⊭ false then
5         path := MakePath(ψ, Parent)
6         if FeasiblePath(path) then
7           return "counterexample path: path "
8         else
9           Preds := RefinePath(path) ∪ Preds
10      else
11        return "program is correct"
    end
```

Fig. 10 Predicate abstraction and refinement loop

Here, we omit the details of a particular algorithm for finding $\varphi_0, \ldots, \varphi_n$ that satisfy the above conditions. We discuss possible alternatives in Sect. 15.7.

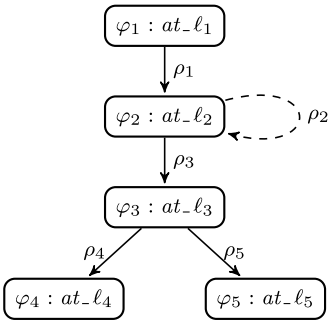
Example 20 As discussed in Example 17, the application of RefinePath on $\rho_1\rho_3\rho_5$ yields a sequence of sets of states that can refine the abstraction to become sufficiently precise at least for dealing with the considered path. □

In our high-level presentation of the algorithm, we leave open many issues for optimization. For example, the precision of the abstraction may be adapted to the different control locations by using different sets of predicates *Preds* for the definition of the over-approximating function α. The corresponding approach is called *lazy abstraction* [39].

Algorithm for Counterexample-Guided Abstraction Refinement

We put together the building blocks described in the previous section into an algorithm AbstRefineLoop that verifies reachability properties using predicate abstraction and its counterexample-guided refinement. See Fig. 10.

Given a program, AbstRefineLoop discovers a proof or a counterexample by repeatedly applying the following steps. First, we compute an over-approximation $\varphi^{\#}_{reach}$ of the set of reachable states using an abstraction function defined w.r.t. the set of predicates *Preds*, which is empty initially. The over-approximation $\varphi^{\#}_{reach}$ is represented by a set of formulas $ReachStates^{\#}$, where each formula represents a set of states. If the set of error states is disjoint from the computed over-approximation, then AbstRefineLoop stops the iteration process and reports that the program is correct. Otherwise, we consider a formula ψ in $ReachStates^{\#}$ that witnesses the intersection with the error states and use ψ in an attempt to refine the abstraction. Refinement is only possible if the discovered intersection is caused by the imprecision of the currently applied abstraction function. We clarify this question by first



$$\varphi_1 = \alpha(\varphi_{init})$$
$$\varphi_2 = post^{\#}(\varphi_1, \rho_1)$$
$$post^{\#}(\varphi_2, \rho_2) \models \varphi_2$$
$$\varphi_3 = post^{\#}(\varphi_2, \rho_3)$$
$$\varphi_4 = post^{\#}(\varphi_3, \rho_4)$$
$$\varphi_5 = post^{\#}(\varphi_3, \rho_5)$$

Fig. 11 Abstract reachability computation with $Preds = \{false, at\_\ell_1, \ldots, at\_\ell_5\}$

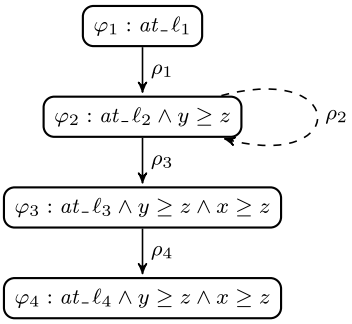
constructing the sequence of program transitions that was traversed during the computation of ψ. This sequence, called *path*, is analyzed using FEASIBLEPATH. If there is a program computation that follows *path*, then ABSTREFINELOOP stops the iteration and reports that *path* is a counterexample. In case *path* is not feasible, we compute a set of predicates that refines the abstraction function by applying an algorithm REFINEPATH on *path*.

We observe that ABSTREFINELOOP never analyzes the same counterexample twice, i.e., the abstraction refinement process using REFINEPATH makes progress at each iteration.

Example 21 We illustrate ABSTREFINELOOP using our example program from Fig. 1. To make the illustration more vivid, we assume that $Preds = \{false, at\_\ell_1, \ldots, at\_\ell_5\}$ is the initial set of predicates, i.e., we anticipate that for proving our example correct we need to keep track of the program counter.

We start the first iteration by applying $ReachStates^{\#}$. The result is the set of formulas $ReachStates^{\#}$ connected by the relation *Parent* as shown in Fig. 11. In this figure, *Parent* is denoted by solid arrows that connect the formulas. We observe that φ_5 has a non-empty intersection with φ_{err}, hence we proceed by setting ψ to φ_5. By applying MAKEPATH we obtain $path = \rho_1\rho_3\rho_5$. At the next step, FEASIBLEPATH reports that this path is not feasible, hence we proceed with the abstraction refinement. REFINEPATH discovers that the predicates $y \geq z$ and $x \geq z$ are sufficient to refine the abstraction such that *path* no longer leads to an error state even under abstraction.

We start the second iteration of ABSTREFINELOOP with the new set of predicates $Preds = \{false, at\_\ell_1, \ldots, at\_\ell_5, y \geq z, x \geq z\}$, which contains the predicates that were discovered during the first iteration. See Fig. 12 for the obtained set $ReachStates^{\#}$ and relation *Parent*. We observe that each formula in $ReachStates^{\#}$ has an empty intersection with φ_{err}, hence ABSTREFINELOOP reports that the program is correct. □



$$\varphi_1 = \alpha(\varphi_{init})$$
$$\varphi_2 = post^{\#}(\varphi_1, \rho_1)$$
$$post^{\#}(\varphi_2, \rho_2) \models \varphi_2$$
$$\varphi_3 = post^{\#}(\varphi_2, \rho_3)$$
$$\varphi_4 = post^{\#}(\varphi_3, \rho_4)$$

Fig. 12 Applying ABSTREACH on the program in Fig. 1 and the set of predicates $Preds = \{false, at\_\ell_1, \ldots, at\_\ell_5, y \geq z, x \geq z\}$

15.6.2 Refinement of Transition Predicate Abstraction

The algorithm TRANSABSTREACH requires a set of transition predicates in order to compute an over-approximation of the transition invariant. Finding the right set of transition predicates that yields a sufficiently precise over-approximation is a difficult task.

Analysis of Counterexample Lassos

We present a notion of lasso-shaped counterexample that is suitable for refining transition predicate abstraction. Such counterexamples consist of a stem and a loop. The stem part represents a sequence of program transitions that leads to a loop in the program, while the loop part is a sequence of program transitions that represents a possible execution through such a loop.

First, we illustrate counterexample lassos using an example.

Example 22 We consider the transition invariant computed in Example 16 by applying TRANSABSTREACH. We assume that $ReachStates^{\#} = \{true\}$ used for this computation was obtained by applying ABSTREACH and that $Parent = \{(true, \rho_1, true)\}$ was obtained as a result.

We observe that $ReachTrans^{\#}$ is not disjunctively well-founded, since it contains $goto(\ell_2, \ell_2)$, which is not well-founded. Similarly to the treatment of counterexamples in predicate-abstraction-based invariant computation, we use *TransParent* to determine the sequence of program transitions that led to the computation of $goto(\ell_2, \ell_2)$, which we call *loop*. We observe that $(true \wedge unchanged(x, y, z), \rho_1, goto(\ell_1, \ell_2)) \in TransParent$, hence the last element of *loop* is the transition ρ_2. Furthermore, since $true \wedge unchanged(x, y, z)$ does not appear in the third position of any element of *TransParent*, we conclude that no more elements appear in *loop*. Now we determine the stem part by applying MAKEPATH on *true*, which is obtained from $true \wedge unchanged(x, y, z)$ by omitting equalities $unchanged(x, y, z)$, and *Parent*. The result is a stem that consists only of ρ_1, which finishes the computation of the counterexample lasso. □

```
    function MakeLasso
    input
      ψ̈ - reachable abstract transition
      Parent - predecessor relation for abstract states
      TransParent - predecessor relation for abstract transitions
    begin
1     loop := empty sequence
2     φ̈′ := ψ̈
3     while exist φ̈ and ρ such that (φ̈, ρ, φ̈′) ∈ TransParent do
4       loop := ρ . loop
5       φ̈′ := φ̈
6     (φ ∧ unchanged(V)) := φ̈
7     stem := MakePath(φ, Parent)
8     return (stem, loop)
    end
```

Fig. 13 Lasso computation

The algorithm MAKELASSO shown in Fig. 13 implements the lasso construction as described in the above example. MAKELASSO proceeds similarly to MAKEPATH and calls it as a sub-routine in line 7 after the loop part is constructed. We detect that the loop construction is finished when there is no predecessor according to *TransParent*. The stem part is constructed using the information collected during the abstract reachability computation and provided as *Parent*. The starting point for the stem computation is obtained using pattern matching in line 6. Here, we exploit how *Worklist* is initialized by TRANSABSTREACH in line 4; see Fig. 5.

Once the lasso counterexample is constructed, we analyse whether the transition predicate abstraction can be refined in order to rule out the discovered counterexample. First, we illustrate this step by using an example.

Example 23 We consider the lasso computed in Example 22, which consists of the stem ρ_1 and the loop ρ_2. We compute the set of states φ that are reachable by applying the stem and obtain

$$\varphi = post(\varphi_{init}, \rho_1) = (at\_\ell_1 \wedge y \geq z).$$

We use this set of states when initializing the relational composition of program transitions along the loop part with

$$at\_\ell_2 \wedge y \geq z \wedge unchanged(x, y, z).$$

The result of the composition—this time without applying transition predicate abstraction—is

$$\begin{aligned}\ddot{\varphi} &= \big(at\_\ell_2 \wedge y \geq z \wedge unchanged(x, y, z)\big) \circ \rho_2 \\ &= \big(goto(\ell_2, \ell_2) \wedge y \geq z \wedge x + 1 \leq y \wedge x' = x + 1 \wedge unchanged(y, z)\big).\end{aligned}$$

Fig. 14 Feasibility of a lasso

```
function FeasibleLasso
input
  ρ1 ... ρn - stem transitions
  ρ'1 ... ρ'm - loop transitions
begin
1   φ := post(φ_init, ρ1 ∘ ... ∘ ρn)
2   φ̈ := (φ ∧ unchanged(V)) ∘ ρ'1 ∘ ... ∘ ρ'm
3   if ¬well-founded(φ̈) then
4     return true
5   else
6     return false
end
```

The obtained relation $\ddot{\varphi}$ is well-founded, which can be easily checked since it is represented by a simple program loop without further nesting or branching statements. A ranking function that witnesses the termination of $\ddot{\varphi}$ is $y - x$. Every time the relation is applied, the value of $y - x$ decreases. Furthermore, $\ddot{\varphi}$ can be applied only if $y - x \geq 0$. We conclude that the discovered counterexample lasso is spurious, i.e., there is no infinite program computation that follows the stem and then repeats the loop part forever. □

The algorithm FeasibleLasso shown in Fig. 14 automates the steps executed in the above example. FeasibleLasso takes as input a lasso obtained by applying MakeLasso and performs a check that the given lasso can yield an infinite computation. The implementation of the predicate *well-founded* is out of the scope of this chapter. There exist efficient algorithms for this task that exploit the lasso shape, e.g., [51].

Finally, we show how transition predicates can be discovered from a spurious counterexample lasso.

Example 24 We consider the feasibility check presented in Example 23. We observe that the following implications were established.

$$post(\varphi_{init}, \rho_1) \models (at\_\ell_1 \wedge y \geq z),$$

$$\big(at\_\ell_2 \wedge y \geq z \wedge unchanged(x, y, z)\big) \circ \rho_2 \models y - x \geq 0 \wedge y' - x' \leq y - x - 1.$$

Hence, to eliminate the spurious counterexample we can use the assertions *true* and $y - x \geq 0 \wedge y' - x' \leq y - x - 1$. When $y - x \geq 0$ and $y' - x' \leq y - x - 1$ are included in the set of transition predicates *TransPreds* used by the abstraction function $\ddot{\alpha}$ the algorithm TransAbstReach will not discover the spurious lasso consisting of ρ_1 and ρ_2 again. Example 14 shows the outcome of applying TransAbstReach when using a refined set of transition predicates *TransPreds*. □

We present an algorithm RefineLasso in Fig. 15 that discovers transition predicates from a spurious counterexample lasso. The algorithm is presented in a declar-

function REFINELASSO
input
$\rho_1 \dots \rho_n$ - stem transitions
$\rho'_1 \dots \rho'_m$ - loop transitions
begin
1 $\varphi_0, \dots, \varphi_n, \ddot{\varphi}_1, \dots, \ddot{\varphi}_m$:= compute such that
2 $(\varphi_{init} \models \varphi_0) \wedge$
3 $(post(\varphi_0, \rho_1) \models \varphi_1) \wedge \dots \wedge (post(\varphi_{n-1}, \rho_n) \models \varphi_n) \wedge$
4 $((\varphi_n \wedge unchanged(V)) \circ \rho'_1 \models \ddot{\varphi}_1) \wedge \dots \wedge (\ddot{\varphi}_{m-1} \circ \rho'_m \models \ddot{\varphi}_m) \wedge$
5 $well\text{-}founded(\ddot{\varphi}_m)$
6 **return** $(\{\varphi_0, \dots, \varphi_n\}, \{\ddot{\varphi}_1, \dots, \ddot{\varphi}_m\})$
end

Fig. 15 Abstraction refinement guided by a lasso

ative way and we omit details of a particular implementation of line 1. There exist efficient implementations for this task that rely on similar techniques to those presented in Sect. 15.7, e.g., [32].

Algorithm for Counterexample-Guided Transition Predicate Abstraction Refinement

We put together the algorithms for the construction and analysis of lasso-shaped counterexamples presented above together with the algorithm for transition predicate abstraction. The resulting algorithm TRANSABSTREFINELOOP can find a disjunctively well-founded transition invariant automatically by automatically discovering an adequate set of transition predicates. See Fig. 16.

TRANSABSTREFINELOOP proceeds in similar steps to ABSTREFINELOOP presented earlier in this section. In fact, we use ABSTREFINELOOP to compute an over-approximation of reachable program states. We start with the empty set of predicates and transition predicates and extend them every time a counterexample lasso is discovered. The counterexample discovery takes place during the computation of a transition invariant using TRANSABSTREACH. If a counterexample lasso is found, its stem part is used to refine the set of predicates *Preds*. The set of additional transition predicates is determined by considering both the stem and the loop parts.

Similarly to abstraction refinement for safety, we observe that TRANSABSTREFINELOOP never analyzes the same counterexample twice, i.e., the abstraction refinement process using REFINELASSO makes progress at each iteration.

15.7 Solving Refinement Constraints for Predicate Abstraction

The algorithm REFINEPATH in Fig. 9 takes as input an infeasible sequence of program transitions $\rho_1 \dots \rho_n$ and computes sets of states $\varphi_0, \dots, \varphi_n$ satisfying the fol-

```
    function TransAbstRefineLoop
    begin
1     Preds := ∅
2     TransPreds := ∅
3     repeat
4       (ReachStates#, Parent) := AbstReach(Preds)
5       (ReachTrans#, TransParent) := TransAbstReach(TransPreds)
6       if exists ψ̈ ∈ ReachTrans# such that ¬well-founded(ψ̈) then
7         (stem, loop) := MakeLasso(ψ̈, Parent, TransParent)
8         if FeasibleLasso(stem, loop) then
9           return "counterexample lasso: stem, loop "
10        else
11          (NewPreds, NewTransPreds) := RefineLasso(stem, loop)
12          Preds := NewPreds ∪ Preds
13          TransPreds := NewTransPreds ∪ TransPreds
14      else
15        return "program terminates"
    end.
```

Fig. 16 Transition predicate abstraction and refinement loop

lowing conditions.

$$\begin{aligned} \varphi_{init} &\models \varphi_0, \\ post(\varphi_0, \rho_1) &\models \varphi_1, \\ &\ldots, \\ post(\varphi_{n-1}, \rho_n) &\models \varphi_n, \\ \varphi_n \wedge \varphi_{err} &\models false. \end{aligned}$$

Since $\rho_1 \ldots \rho_n$ is infeasible, the above conditions are satisfiable. In general, several solutions may exist. We describe how the least, the greatest, and an intermediate solution can be computed.

15.7.1 Least Solution

We obtain the least solution by applying the post-condition function in the following way.

$$\begin{aligned} \varphi_0 &= \varphi_{init}, \\ \varphi_1 &= post(\varphi_0, \rho_1), \\ &\ldots, \\ \varphi_n &= post(\varphi_{n-1}, \rho_n). \end{aligned} \tag{22}$$

Note that the least solution ensures that for each $1 \leq i \leq n$ we have

$$\varphi_i = post(\varphi_{init}, \rho_1 \circ \cdots \circ \rho_i),$$

and guarantees that $\varphi_n \wedge \varphi_{err} \models false$.

Sometimes the least solution is not useful for refining the abstraction, since the resulting abstraction is too precise. As a result, the iteration in AbstRefineLoop may not terminate as the abstract reachability computation is almost equivalent to the reachability computation without abstraction.

Example 25 We illustrate how a least solution is computed using the example program shown in Fig. 1.

Let $\rho_1\rho_3\rho_5$ be a counterexample path discovered by AbstRefineLoop. For this path, we obtain the following least solution of the constraints defined by RefinePath.

$$\begin{aligned}
\varphi_0 &= \varphi_{init} &&= at_\ell_1,\\
\varphi_1 &= post(\varphi_0, \rho_1) &&= (at_\ell_2 \wedge y \geq z),\\
\varphi_2 &= post(\varphi_1, \rho_3) &&= (at_\ell_3 \wedge y \geq z \wedge x \geq y),\\
\varphi_3 &= post(\varphi_2, \rho_5) &&= false.
\end{aligned}$$

The obtained refinement will ensure that the path $\rho_2\rho_3\rho_5$ will not be considered a counterexample during subsequent iterations of the refinement loop in AbstRefineLoop. □

15.7.2 Greatest Solution

First, we define an auxiliary *weakest pre-condition* function *wp* as follows. Let φ be a formula over V and let ρ be a formula over V and V'. Then, we define:

$$wp(\varphi, \rho) = \forall V' : \rho \rightarrow \varphi\big[V'/V\big]. \tag{23}$$

For example, the transition ρ_2 from Fig. 1 results in the following weakest precondition.

$$\begin{aligned}
&wp(at_\ell_2 \wedge x \geq z, \rho_2)\\
&\quad = \forall V' : pc = \ell_2 \wedge x + 1 \leq y \wedge x' = x + 1 \wedge y' = y \wedge z' = z \wedge pc' = \ell_2\\
&\qquad \rightarrow pc' = \ell_2 \wedge x' \geq z\\
&\quad = \neg\big(\exists V' : pc = \ell_2 \wedge x + 1 \leq y \wedge x' = x + 1 \wedge y' = y \wedge z' = z \wedge pc' = \ell_2 \wedge\\
&\qquad \neg\big(pc' = \ell_2 \wedge x' \geq z\big)\big)\\
&\quad = \neg\big(\exists V' : pc = \ell_2 \wedge x + 1 \leq y \wedge \neg(\ell_2 = \ell_2 \wedge x + 1 \geq z)\big)
\end{aligned}$$

$$= (pc = \ell_2 \wedge x + 1 \leq y \rightarrow \ell_2 = \ell_2 \wedge x + 1 \geq z)$$
$$= (at\_\ell_2 \wedge x + 1 \leq y \rightarrow x + 1 \geq z).$$

We obtain the greatest solution of the refinement constraints for a given counterexample path as follows.

$$\begin{aligned} \varphi_n &= \neg\varphi_{err}, \\ \varphi_{n-1} &= wp(\varphi_n, \rho_n), \\ &\ldots, \\ \varphi_0 &= wp(\varphi_1, \rho_1). \end{aligned} \tag{24}$$

That is, the greatest solution is computed incrementally by traversing the counterexample path backwards.

Similarly to the least solution, sometimes the greatest solution is not useful for refining the abstraction, since the resulting abstraction is too coarse. As a result, the iteration in AbstRefineLoop may not terminate as the abstract reachability computation is almost equivalent to the backward reachability computation without abstraction that expands the set of states definitely leading to an error state.

Example 26 We illustrate how a greatest solution is computed using an example program shown in Fig. 1.

Let $\rho_1\rho_3\rho_5$ be a counterexample path discovered by AbstRefineLoop. For this path, we obtain the following greatest solution of the constraints in RefinePath.

$$\begin{aligned} \varphi_3 &= \neg\varphi_{err} &&= \neg at\_\ell_5, \\ \varphi_2 &= wp(\varphi_3, \rho_5) &&= (at\_\ell_3 \rightarrow x \geq z), \\ \varphi_1 &= wp(\varphi_2, \rho_3) &&= (at\_\ell_2 \wedge x \geq y \rightarrow x \geq z), \\ \varphi_0 &= wp(\varphi_1, \rho_1) &&= true. \end{aligned}$$

Again, the obtained refinement will result in the discovery of the counterexample path $\rho_1\rho_3\rho_5$ during the next iteration of AbstRefineLoop, as witnessed by the following validities.

$$\begin{aligned} &\varphi_{init} \models \varphi_0, \\ &post(\varphi_0, \rho_1) = (at\_\ell_2 \wedge y \geq z) \models \varphi_1, \\ &post(\varphi_1, \rho_3) = (at\_\ell_3 \wedge x \geq y \wedge x \geq z) \models \varphi_2, \\ &post(\varphi_2, \rho_5) = false \models \varphi_3. \end{aligned}$$

We observe that the reachability computation using refined abstraction does not reach any error states along the path $\rho_1\rho_3\rho_5$. □

15.7.3 Intermediate Solution Using Interpolation

We illustrate how an intermediate solution can be computed by a technique called interpolation [25, 38]. Interpolation takes as input two mutually unsatisfiable formulas φ_1 and φ_2, i.e., $\varphi_1 \wedge \varphi_2 \models \mathit{false}$, and returns an *interpolant*, a formula φ such that (i) φ is expressed over common symbols of φ_1 and φ_2, (ii) $\varphi_1 \models \varphi$, and (iii) $\varphi \wedge \varphi_2 \models \mathit{false}$. Let *inter* be an interpolation function such that $\mathit{inter}(\varphi_1, \varphi_2)$ is an interpolant for φ_1 and φ_2.

The following sequence of interpolation computations can be used to find a solution for constraints defined by RefinePath.

$$\begin{aligned}
\varphi_0 &= \mathit{inter}\big(\varphi_{\mathit{init}}, (\rho_1 \circ \cdots \circ \rho_n) \wedge \varphi_{\mathit{err}}\big[V'/V\big]\big),\\
\varphi_1 &= \mathit{inter}\big(\mathit{post}(\varphi_0, \rho_1), (\rho_2 \circ \cdots \circ \rho_n) \wedge \varphi_{\mathit{err}}\big[V'/V\big]\big),\\
&\ldots,\\
\varphi_{n-1} &= \mathit{inter}\big(\mathit{post}(\varphi_{n-2}, \rho_{n-1}), \rho_n \wedge \varphi_{\mathit{err}}\big[V'/V\big]\big),\\
\varphi_n &= \mathit{inter}\big(\mathit{post}(\varphi_{n-1}, \rho_n), \varphi_{\mathit{err}}\big[V'/V\big]\big).
\end{aligned} \tag{25}$$

Intermediate solutions can avoid the deficiencies of least and greatest solutions described above, although they still do not guarantee convergence of the abstraction refinement loop.

Example 27 We illustrate how an intermediate solution is computed using the example program shown in Fig. 1.

Let $\rho_1\rho_3\rho_5$ be a counterexample path discovered by AbstRefineLoop. For this path, we obtain the following intermediate solution of the constraints in RefinePath.

$$\begin{aligned}
\varphi_0 &= \mathit{inter}\big(\varphi_{\mathit{init}}, (\rho_1 \circ \rho_3 \circ \rho_5) \wedge \varphi_{\mathit{err}}\big[V'/V\big]\big) &&= \mathit{true},\\
\varphi_1 &= \mathit{inter}\big(\mathit{post}(\varphi_0, \rho_1), (\rho_3 \circ \rho_5) \wedge \varphi_{\mathit{err}}\big[V'/V\big]\big) &&= y \geq z,\\
\varphi_2 &= \mathit{inter}\big(\mathit{post}(\varphi_1, \rho_3), \rho_5 \wedge \varphi_{\mathit{err}}\big[V'/V\big]\big) &&= x \geq z,\\
\varphi_3 &= \mathit{inter}\big(\mathit{post}(\varphi_2, \rho_5), \varphi_{\mathit{err}}\big[V'/V\big]\big) &&= \mathit{false}.
\end{aligned}$$

The following validities show that $\rho_1\rho_3\rho_5$ will not be considered a counterexample during subsequent refinement iterations.

$$\begin{aligned}
&\varphi_{\mathit{init}} \models \varphi_0,\\
&\mathit{post}(\varphi_0, \rho_1) = (\mathit{at_}\ell_2 \wedge y \geq z) \models \varphi_1,\\
&\mathit{post}(\varphi_1, \rho_3) = (\mathit{at_}\ell_3 \wedge x \geq y \wedge y \geq z) \models \varphi_2,\\
&\mathit{post}(\varphi_2, \rho_5) = \mathit{false} \models \varphi_3.
\end{aligned}$$

□

15.8 Tools

We have presented the base algorithm for predicate abstraction and transition predicate abstraction. Practical tools introduce a variety of optimizations of the base algorithm.

Predicate Abstraction

SLAM [5], BLAST [38, 39], Magic [15], Murφ [27], and SatAbs [18] implement different levels of precision, ranging from Cartesian to full Boolean predicate abstraction [3]. CPAChecker [10], F-Soft [41], and UFO [1] integrate predicate abstraction with data flow analysis and abstract interpretation. Synergy [33] and Yogi [50] integrate predicate abstraction with under-approximation based on dynamic execution. ARMC [54] implements Cartesian predicate abstraction and uses constraint-based interpolation to discover predicates. SLAB [28] implements the refinement of an abstract transition system in a top-down way. Impact [49] and Wolverine [45] resort to a particular form of predicate abstraction where each refinement step adds a single predicate. Ultimate Automizer [36] uses predicates to construct a proof in the form of a finite automaton that approximates the language of program traces.

Arrays and Heaps

BLAST [43], Indexed Predicate Abstraction [46], and universally quantified Horn solver [12] compute universally quantified array invariants with predicate abstraction in order to deal with the ranges of array indices and properties of values stored in arrays. Bohne [55, 56] verifies complex data structures that are implemented on the heap (modeled as a graph) by inferring node predicates in the style of TVLA [58].

Beyond Procedural Programs

HSF [32] relies on predicate abstraction to solve recursive Horn constraints, which serves as a back-end solver for proving temporal properties of programs with procedures, multi-threaded programs, and higher-order functional programs. Threader [34, 35] relies on predicate abstraction to compute rely-guarantee and Owicki-Gries proofs for multi-threaded programs. Liquid Types [57] uses a form of predicate abstraction in the style of Houdini [29] to infer refinement types for proving safety of higher-order functional programs.

Beyond Safety

ARMC [54] uses transition predicate abstraction as described in [53] to prove termination and other liveness properties. Terminator [20, 21] reduces transition predicate abstraction to predicate abstraction via a syntactic transformation of the program in order to prove termination of systems code. T2 [13, 22] computes transition invariants using the same techniques as Impact [49]. LoopFrog computes transition invariants to analyze termination of programs using a bit-level semantics. LTA [48] uses algorithmic-learning-based techniques for the generation of transition predicates. CTA [44] computes ‘compositional’ transition invariants. Ultimate

Automizer [36] uses transition predicates to construct a proof in the form of a finite Büchi automaton that approximates the language of infinite program traces. Several tools including AProVe [14] and ACL2 [16] use the size-change principle [47], whose formal connection to transition predicate abstraction (without refinement) is studied in [37]. An explanation for why transition predicate abstraction works for termination analysis is given in the abstract interpretation framework in [24]. HSF [32] relies on transition predicate abstraction in combination with abstract inference to find well-founded models for Horn constraints.

Beyond Verification

Existentially quantified Horn solver [7] uses predicate abstraction to discover witness existential quantification in Horn constraints and to synthesize winning strategies for LTL games [6].

15.9 Conclusion

We have presented an automated over-approximation technique called predicate abstraction and we have shown how it can be applied for proving non-reachability and termination, the two base properties to which many correctness specifications for programs can be reduced. The implementation of predicate abstraction relies on decision procedures for entailment. Given a verification problem, an adequate set of predicates can be discovered automatically, namely by exploring spurious counterexamples.

Our presentation aims at the basic principles of predicate abstraction. It leaves uncovered the variations of predicate abstraction that are studied in the literature and implemented in tools. We refer to [42] for a survey. Chapter 13 of this Handbook [26] shows how the process of computing an over-approximation of a given transition relation for a given set of predicates can be decoupled from the fixpoint computation. Chapter 16 of this Handbook [9] shows how predicate abstraction can be combined with data flow analysis, thus only requiring decision procedure calls for intricate reasoning that is difficult to support in classical data flow domains.

References

1. Albarghouthi, A., Li, Y., Gurfinkel, A., Chechik, M.: UFO: a framework for abstraction- and interpolation-based software verification. In: Madhusudan, P., Seshia, S.A. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 7358, pp. 672–678. Springer, Heidelberg (2012)
2. Ball, T., Majumdar, R., Millstein, T.D., Rajamani, S.K.: Automatic predicate abstraction of C programs. In: Burke, M., Soffa, M.L. (eds.) Conf. on Programming Language Design and Implementation (PLDI), pp. 203–213. ACM, New York (2001)
3. Ball, T., Podelski, A., Rajamani, S.K.: Boolean and Cartesian abstraction for model checking C programs. Int. J. Softw. Tools Technol. Transf. **5**(1), 49–58 (2003)

4. Ball, T., Rajamani, S.K.: Bebop: a symbolic model checker for boolean programs. In: Havelund, K., Penix, J., Visser, W. (eds.) Intl. Workshop on Model Checking Software (SPIN). LNCS, vol. 1885, pp. 113–130. Springer, Heidelberg (2000)
5. Ball, T., Rajamani, S.K.: The SLAM project: debugging system software via static analysis. In: Launchbury, J., Mitchell, J.C. (eds.) Symp. on Principles of Programming Languages (POPL), pp. 1–3. ACM, New York (2002)
6. Beyene, T.A., Chaudhuri, S., Popeea, C., Rybalchenko, A.: A constraint-based approach to solving games on infinite graphs. In: Jagannathan, S., Sewell, P. (eds.) Symp. on Principles of Programming Languages (POPL), pp. 221–234. ACM, New York (2014)
7. Beyene, T.A., Popeea, C., Rybalchenko, A.: Solving existentially quantified Horn clauses. In: Sharygina, N., Veith, H. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 8044, pp. 869–882. Springer, Heidelberg (2013)
8. Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M.E., Sebastiani, R.: Software model checking via large-block encoding. In: Formal Methods in Computer Aided Design (FMCAD), pp. 25–32. IEEE, Piscataway (2009)
9. Beyer, D., Gulwani, S., Schmidt, D.A.: Combining model checking and data-flow analysis. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
10. Beyer, D., Keremoglu, M.E.: CPAchecker: a tool for configurable software verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg (2011)
11. Beyer, D., Keremoglu, M.E., Wendler, P.: Predicate abstraction with adjustable-block encoding. In: Bloem, R., Sharygina, N. (eds.) Formal Methods in Computer Aided Design (FMCAD), pp. 189–197. IEEE, Piscataway (2010)
12. Bjørner, N., McMillan, K.L., Rybalchenko, A.: On solving universally quantified Horn clauses. In: Logozzo, F., Fähndrich, M. (eds.) Intl. Symp. on Static Analysis (SAS). LNCS, vol. 7935, pp. 105–125. Springer, Heidelberg (2013)
13. Brockschmidt, M., Cook, B., Fuhs, C.: Better termination proving through cooperation. In: Sharygina, N., Veith, H. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 8044, pp. 413–429. Springer, Heidelberg (2013)
14. Brockschmidt, M., Musiol, R., Otto, C., Giesl, J.: Automated termination proofs for Java programs with cyclic data. In: Madhusudan, P., Seshia, S.A. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 7358, pp. 105–122. Springer, Heidelberg (2012)
15. Chaki, S., Clarke, E.M., Groce, A., Jha, S., Veith, H.: Modular verification of software components in C. In: Clarke, L.A., Dillon, L., Tichy, W.F. (eds.) Intl. Conf. on Software Engineering (ICSE), pp. 385–395. IEEE, Piscataway (2003)
16. Chamarthi, H.R., Dillinger, P.C., Manolios, P., Vroon, D.: The ACL2 Sedan theorem proving system. In: Abdulla, P.A., Leino, K.R.M. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 6605, pp. 291–295. Springer, Heidelberg (2011)
17. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000)
18. Clarke, E.M., Kroening, D., Sharygina, N., Yorav, K.: SATABS: SAT-based predicate abstraction for ANSI-C. In: Halbwachs, N., Zuck, L.D. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 3440, pp. 570–574. Springer, Heidelberg (2005)
19. Clarke, E.M., Kurshan, R.P., Veith, H.: The localization reduction and counterexample-guided abstraction refinement. In: Manna, Z., Peled, D.A. (eds.) Essays in Memory of Amir Pnueli. LNCS, vol. 6200, pp. 61–71. Springer, Heidelberg (2010)
20. Cook, B., Podelski, A., Rybalchenko, A.: Termination proofs for systems code. In: Schwartzbach, M.I., Ball, T. (eds.) Conf. on Programming Language Design and Implementation (PLDI), pp. 415–426. ACM, New York (2006)

21. Cook, B., Podelski, A., Rybalchenko, A.: Terminator: beyond safety. In: Ball, T., Jones, R.B. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 4144, pp. 415–418. Springer, Heidelberg (2006)
22. Cook, B., See, A., Zuleger, F.: Ramsey vs. lexicographic termination proving. In: Piterman, N., Smolka, S.A. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 7795, pp. 47–61. Springer, Heidelberg (2013)
23. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Graham, R.M., Harrison, M.A., Sethi, R. (eds.) Symp. on Principles of Programming Languages (POPL), pp. 238–252. ACM, New York (1977)
24. Cousot, P., Cousot, R.: An abstract interpretation framework for termination. In: Field, J., Hicks, M. (eds.) Symp. on Principles of Programming Languages (POPL), pp. 245–258. ACM, New York (2012)
25. Craig, W.: Linear reasoning. A new form of the Herbrand-Gentzen theorem. J. Symb. Log. **22**(3), 250–268 (1957)
26. Dams, D., Grumberg, O.: Abstraction and abstraction refinement. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
27. Das, S., Dill, D.L., Park, S.: Experience with predicate abstraction. In: Halbwachs, N., Peled, D.A. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 1633, pp. 160–171. Springer, Heidelberg (1999)
28. Dräger, K., Kupriyanov, A., Finkbeiner, B., Wehrheim, H.: SLAB: a certifying model checker for infinite-state concurrent systems. In: Esparza, J., Majumdar, R. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 6015, pp. 271–274. Springer, Heidelberg (2010)
29. Flanagan, C., Leino, K.R.M.: Houdini, an annotation assistant for ESC/Java. In: Oliveira, J.N., Zave, P. (eds.) Intl. Symp. on Formal Methods Europe (FME). LNCS, vol. 2021, pp. 500–517. Springer, Heidelberg (2001)
30. Floyd, R.W.: Assigning meanings to programs. In: Schwartz, J.T. (ed.) Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics, pp. 19–32. AMS, Providence (1967)
31. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
32. Grebenshchikov, S., Lopes, N.P., Popeea, C., Rybalchenko, A.: Synthesizing software verifiers from proof rules. In: Vitek, J., Lin, H., Tip, F. (eds.) Conf. on Programming Language Design and Implementation (PLDI), pp. 405–416. ACM, New York (2012)
33. Gulavani, B.S., Henzinger, T.A., Kannan, Y., Nori, A.V., Rajamani, S.K.: SYNERGY: a new algorithm for property checking. In: Young, M., Devanbu, P.T. (eds.) Intl. Symp. on Foundations of Software Engineering (FSE), pp. 117–127. ACM, New York (2006)
34. Gupta, A., Popeea, C., Rybalchenko, A.: Predicate abstraction and refinement for verifying multi-threaded programs. In: Ball, T., Sagiv, M. (eds.) Symp. on Principles of Programming Languages (POPL), pp. 331–344. ACM, New York (2011)
35. Gupta, A., Popeea, C., Rybalchenko, A.: Threader: a constraint-based verifier for multi-threaded programs. In: Gopalakrishnan, G., Qadeer, S. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 6806, pp. 412–417. Springer, Heidelberg (2011)
36. Heizmann, M., Hoenicke, J., Podelski, A.: Nested interpolants. In: Hermenegildo, M.V., Palsberg, J. (eds.) Symp. on Principles of Programming Languages (POPL), pp. 471–482. ACM, New York (2010)
37. Heizmann, M., Jones, N.D., Podelski, A.: Size-change termination and transition invariants. In: Cousot, R., Martel, M. (eds.) Intl. Symp. on Static Analysis (SAS). LNCS, vol. 6337, pp. 22–50. Springer, Heidelberg (2010)
38. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: Jones, N.D., Leroy, X. (eds.) Symp. on Principles of Programming Languages (POPL), pp. 232–244. ACM, New York (2004)

39. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Launchbury, J., Mitchell, J.C. (eds.) Symp. on Principles of Programming Languages (POPL), pp. 58–70. ACM, New York (2002)
40. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
41. Ivancic, F., Shlyakhter, I., Gupta, A., Ganai, M.K.: Model checking C programs using F-SOFT. In: International Conference on Computer Design (ICCD), pp. 297–308. IEEE, Piscataway (2005)
42. Jhala, R., Majumdar, R.: Software model checking. ACM Comput. Surv. **41**(4), 21:1–21:54 (2009)
43. Jhala, R., McMillan, K.L.: Array abstractions from proofs. In: Damm, W., Hermanns, H. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 4590, pp. 193–206. Springer, Heidelberg (2007)
44. Kroening, D., Sharygina, N., Tsitovich, A., Wintersteiger, C.M.: Termination analysis with compositional transition invariants. In: Touili, T., Cook, B., Jackson, P. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 6174, pp. 89–103. Springer, Heidelberg (2010)
45. Kroening, D., Weissenbacher, G.: Interpolation-based software verification with Wolverine. In: Gopalakrishnan, G., Qadeer, S. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 6806, pp. 573–578. Springer, Heidelberg (2011)
46. Lahiri, S.K., Bryant, R.E.: Indexed predicate discovery for unbounded system verification. In: Alur, R., Peled, D.A. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 3114, pp. 135–147. Springer, Heidelberg (2004)
47. Lee, C.S., Jones, N.D., Ben-Amram, A.M.: The size-change principle for program termination. In: Hankin, C., Schmidt, D. (eds.) Symp. on Principles of Programming Languages (POPL), pp. 81–92. ACM, New York (2001)
48. Lee, W., Wang, B., Yi, K.: Termination analysis with algorithmic learning. In: Madhusudan, P., Seshia, S.A. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 7358, pp. 88–104. Springer, Heidelberg (2012)
49. McMillan, K.L.: Lazy abstraction with interpolants. In: Ball, T., Jones, R.B. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 4144, pp. 123–136. Springer, Heidelberg (2006)
50. Nori, A.V., Rajamani, S.K., Tetali, S., Thakur, A.V.: The Yogi project: software property checking via static analysis and testing. In: Kowalewski, S., Philippou, A. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 5505, pp. 178–181. Springer, Heidelberg (2009)
51. Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In: Steffen, B., Levi, G. (eds.) Intl. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI). LNCS, vol. 2937, pp. 239–251. Springer, Heidelberg (2004)
52. Podelski, A., Rybalchenko, A.: Transition invariants. In: Symp. on Logic in Computer Science (LICS), pp. 32–41. IEEE, Piscataway (2004)
53. Podelski, A., Rybalchenko, A.: Transition predicate abstraction and fair termination. In: Palsberg, J., Abadi, M. (eds.) Symp. on Principles of Programming Languages (POPL), pp. 132–144. ACM, New York (2005)
54. Podelski, A., Rybalchenko, A.A.: The logical choice for software model checking with abstraction refinement. In: Hanus, M. (ed.) Practical Aspects of Declarative Languages (PADL). LNCS, vol. 4354, pp. 245–259. Springer, Heidelberg (2007)
55. Podelski, A., Wies, T.: Boolean heaps. In: Hankin, C., Siveroni, I. (eds.) Intl. Symp. on Static Analysis (SAS). LNCS, vol. 3672, pp. 268–283. Springer, Heidelberg (2005)
56. Podelski, A., Wies, T.: Counterexample-guided focus. In: Hermenegildo, M.V., Palsberg, J. (eds.) Symp. on Principles of Programming Languages (POPL), pp. 249–260. ACM, New York (2010)

57. Rondon, P.M., Kawaguchi, M., Jhala, R.: Liquid types. In: Gupta, R., Amarasinghe, S.P. (eds.) Conf. on Programming Language Design and Implementation (PLDI), pp. 159–169. ACM, New York (2008)
58. Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. **24**(3), 217–298 (2002)

Chapter 16
Combining Model Checking and Data-Flow Analysis

Dirk Beyer, Sumit Gulwani, and David A. Schmidt

Abstract Until recently, model checking and data-flow analysis—two traditional approaches to software verification—were used independently and in isolation for solving similar problems. Theoretically, the two different approaches are equivalent; they are two different ways to compute the same solution to a problem. In recent years, new practical approaches have shown how to combine the approaches and how to make them benefit from each other—model-checking techniques can make data-flow analyses more precise, and data-flow-analysis techniques can make model checking more efficient. This chapter starts by discussing the relationship (differences and similarities) between type checking, data-flow analysis, and model checking. Then we define algorithms for data-flow analysis and model checking in the same formal setting, called configurable program analysis. This identifies key differences that make us call an algorithm a "model-checking" algorithm or a "data-flow-analysis" algorithm. We illustrate the effect of using different algorithms for running certain classic example analyses and point out the reason for one algorithm being "better" than the other. The chapter presents combined verification techniques in the framework of configurable program analysis, in order to emphasize techniques used in data-flow analysis and in model checking. Besides the iterative algorithm that is used to illustrate the similarities and differences between data-flow analysis and model checking, we discuss different algorithmic approaches for constructing program invariants. To show that the border between data-flow analysis and model checking is blurring and disappearing, we also discuss directions in tool implementations for combined verification approaches.

D. Beyer (✉)
Ludwig-Maximilians-Universität München, Munich, Germany
e-mail: dirk.beyer@sosy-lab.org

S. Gulwani
Microsoft Research, Redmond, WA, USA

D.A. Schmidt
Kansas State University, Manhattan, KS, USA

E.M. Clarke et al. (eds.), *Handbook of Model Checking*,
DOI 10.1007/978-3-319-10575-8_16

16.1 Introduction

In the context of software verification, model checking is considered a semi-decidable, exhaustive, and precise analysis of an abstract model of a program, whereas data-flow analysis is considered a terminating, imprecise abstract interpretation of a concrete model of a program.

For example, to validate a safety property, abstraction-refinement-based model checking creates an abstract model of the program and precisely analyzes every reachable abstract state for the property, repeatedly refining and rechecking the model until validation is achieved, whereas a classic data-flow analysis computes abstract values of the states that arrive at the program locations of the concrete program, repeatedly computing and combining the abstract values until convergence at all program locations is achieved. Classic data-flow analyses are efficient (as required for compiler optimization) at the cost of precision. Model checkers aim at being precise (as required for proof construction) at the cost of efficiency.

Precisely defining the difference between model checking and data-flow analysis is not easy, and indeed the two approaches have been proven to be "the same" in that each can be coded in the framework of the other. This chapter illustrates why the two approaches are theoretically equivalent—they are two fashions of computing the same solution. In practice, the two approaches are extremes in a spectrum of many possible algorithms, and the spectrum can be defined by a few parameters that describe the different implementation techniques. As soon as we set the parameters differently from the extremes that define the two approaches, we see how new combinations are possible. While in most of the chapter we assume that the popular, iteration-based algorithm is used, we later also provide a comparative overview of other algorithmic approaches for constructing program invariants.

In this chapter, we restrict ourselves to verifying safety properties of software.

16.2 General Considerations

For background, we compare and contrast three techniques that are widely used for static (pre-execution) program validation: type checking, data-flow analysis, and model checking. These techniques come with different representations of *program*, *property*, and *analysis algorithm*. We describe here the commonly used versions, but with small extensions it is possible for each technique to express the other two [108].

16.2.1 Type Checking

Type checking is an analysis of a program's syntax tree that attaches properties ("types") to the phrases that are embedded in the tree. Types might be primitive (*int*, *float*, *string*, *void*) or compound (*int*[ ], *string* × *float*, {"`name`" : *string*, "`age`" : *int*})

or phrase-type (*command*(*int*), *declaration*(*ident*, *float*)). For example, the C command `float x = y + 1.5` might be parsed and type checked like this:

$$\left(\texttt{float x} = \left(\texttt{y}^{int} + 1.5^{float}\right)^{float}\right)^{declaration(\texttt{x}, float)}$$

provided that `y`'s declaration was typed by *declaration*(`y`, *int*).

Type checking can validate safety properties ("a well-typed program cannot go wrong at execution") and can help a compiler generate target code.

Program Representation. A program's syntax tree (parse tree) is used for type checking. The tree is often accompanied by a symbol table that holds typings of free (global) variables.

Property Representation. There is no firm designation as to what types are, but a type should have semantic significance. Types are typically defined inductively. The earlier example used types derived from this grammar:

$$\begin{array}{ll} p : \mathit{PhraseType} & a : \mathit{ExpressionType} \\ \quad p ::= \mathit{command}(a) \mid \mathit{declaration}(\mathit{ident}, a) & \\ \quad a ::= \mathit{int} \mid \mathit{float} \mid a[\,] & \end{array}$$

The type language resembles a propositional logic, where primitive types (*int*, *string*) define the primitive propositions and compound types (*command*(*a*), *declaration*(*ident*, *a*)) define the compound propositions. Data structures, such as arrays, tuples, structs, and function closures, are annotated with compound types.

The "type logic" need not be a mere propositional logic. Languages that support templates or parametric polymorphism, e.g., Standard ML [43], include Prolog-style logical variables in the syntax of types; the logical variables are placeholders for types that are inserted later, or they are understood as universally quantified variables. For example, $\alpha \to (\alpha \times \alpha)$ is a typing of this function definition:

$$\texttt{define f(x)} = \texttt{makePair(x, x)}.$$

The occurrences of α are placeholders that can be filled later, e.g., as in $f(\texttt{1.5})$ (α is replaced by *float*) or $f(\texttt{"hello"})$ (α is replaced by *string*). Indeed, the type can be read as the predicate-logic formula $\forall\alpha(\alpha \to (\alpha \wedge \alpha))$ [93].

At the other extreme, the type language can be an ad hoc collection of labels, provided there is some significance as to how the labels annotate the syntax tree. An example is *value numbering*, where each expression node is annotated by the set of expression nodes in the tree whose run-time values will equal the present node's [101].

Analysis Algorithm. A finite (usually, left-to-right, one- or two-pass) tree-traversal algorithm attaches types to the nodes of the syntax tree. In the language of Knuthian attribute grammars [82], properties that are *inherited* are carried from parent nodes to child nodes for further computation, and properties that are *synthesized* are com-

municated from child nodes to parent nodes. In the first example in this section, variable `y`'s type is inherited information that is passed to the phrase (`y + 1.5`), which synthesizes the type *float*. The algorithm for attaching properties can be written with attribute-grammar equations [86] or inference rules [97].

The equations (or rules) are meant to be deterministic, but some program phrases might be annotated with multiple acceptable choices (e.g., `2` : *int* and also `2` : *float*). In this case, an ordering, $\leq$, as in $int \leq float$, lets one deduce a most precise property for a phrase. This concept, called *subtyping* [1], is central to type checking for object-oriented languages. Using logical variables, ML's Algorithm W [43] deduces a most general typing for an ML program that can be correctly typed in multiple ways.

Extensions. If the typing language is complex enough, it can express any or all semantic properties of a program, e.g., a phrase's "type" might be its compiled code or it might be the input-output function that the phrase denotes! (The former is called a *syntax-directed translation* [2] and the latter is the program's *denotational semantics* [106].)

When the typing language is a logic, a type checker reads a syntax tree as a "proof" of a "proposition," namely, the program's type—the type checker does proof checking [95].[1]

The algorithm that attaches properties to the program tree might repeatedly traverse the tree and compute the types attached to the program locations until a convergence is achieved. (The iteration is a least-fixed-point computation, requires that the property language is partially ordered, and uses a join (union) operation to refine types.) This algorithm leads to the next analysis form, because it is a *structured data-flow analysis* [2].

Further, if the type language is a temporal logic, the iterative traversal computes the temporal properties that are valid at each node of the tree—from here, it is a small step to branching-time model checking on Kripke structures. Further examples of fixed-point computation on parse trees are discussed in the literature [40].

16.2.2 Data-Flow Analysis

Data-flow analysis predicts the "flow" of information through the locations of a program. The flow can be computed either forward or backward and is often set-like, e.g., predicting the set of arithmetic expressions that have been previously evaluated at a program location, or the set of variables that will be needed for future computation, or the set of variables that definitely have constant values, or the set of aliased pointers. Because the analysis over-approximates a program's possible execution sequences, its results are imprecise.

The information gathered by a data-flow analysis can be used to validate safety properties or to help a compiler generate efficient target code. For example, if a

[1] The connection between model checking and theorem proving is discussed in Chap. 20.

constant-propagation analysis calculates that a variable has a known constant value at a program location, a constant-load instruction can replace the storage-lookup instruction.

Program Representation. A program is portrayed as a *directed graph*, whose nodes represent program locations. An edge connects two nodes if execution can transfer control from one location to the next; the initial node represents the program entry; final nodes represent the program exits. The edges are labeled by the primitive action (assignment, assume operation) that transfers control—the directed graph is a *control-flow automaton* (CFA),[2] which displays the program's operations and its semantics of control. Figure 1 shows an example C function (a) together with its CFA (b). The CFA might be further condensed [9] or unrolled ("expanded") [115] so that more precise properties can be computed for its nodes.

Property Representation. Data-flow analysis annotates the CFA's locations with properties, which are usually sets that have semantic significance. A program location's property set might indicate the variables or expressions whose values were transferred along a control path to the program location [78]. For example, an available-expressions analysis predicts which arithmetic expressions (that appear in the program's text) will definitely be evaluated and be ready for use at subsequent program locations. The *property language* for available-expressions analysis is the collection of all subsets of expressions that appear in the program.

Another example is a constant-propagation analysis, which predicts which variables possess specific, constant values at the program locations. The property language consists of sets of the form $\{(\mathtt{x}_0, c_0), \ldots, (\mathtt{x}_n, c_n)\}$, where each $\mathtt{x}_i$ is a variable name in the program, all $\mathtt{x}_i$ are unique, and c_i is a fixed, constant value (e.g., 1.5 or 2) *or* the "unspecific value" $\top$, which indicates that $\mathtt{x}_i$ is defined but cannot be validated as having a constant value.

Analysis Algorithm. An iterative algorithm computes the properties that annotate the program locations. Starting from an initial configuration, where some program locations are annotated with input information, the algorithm propagates the properties along the CFA's edges. Recall that each edge from program location l to program location l' is labeled by an action. The semantics of the action is defined by a *transfer function* $f_{l \to l'}$, which defines how properties are updated when they traverse the edge [38]. There are two important versions of the analysis algorithm [2, 77, 78]:

1. *Maximal Fixed Point (MFP)*: Properties for each program location are computed directly on the CFA. The information computed for program location l' is defined

[2] Although in principle equivalent to the classical control-flow graphs [2], assigning the program operations to the edges is more compatible with the model-checking view (cf. the more detailed discussion by Steffen [113, 114]). The notion of control-flow automata is meanwhile established as a standard representation for programs (cf. the implementation in BLAST [13] and other verifiers).

by an equation of the following form [78]:

$$propertyAtNode(l') = \bigcup_{l \in pred(l')} f_{l \to l'}(propertyAtNode(l))$$

where $pred(l')$ is the set of all predecessor locations for location l' in the CFA, and $f_{l \to l'}$ is the transfer function. The equations for the CFA's program locations are initialized to a minimal value, e.g., {}, and are iterated until they stabilize. To ensure finite convergence, the property language can be made finite and partially ordered (say, by subset, $\subseteq$). More generally, the property language can be a lattice of finite height [38].

2. *Meet Over all Paths (MOP)*: The iterative analysis algorithm enumerates the *paths within the CFA*. Properties are computed for all program locations on each individual path, and the results of all the analyses on all paths are joined. Since a program might have an infinite number of paths, path generation must be made convergent, say by bounding the expansion of loops[3] and procedure calls.

This example shows the distinction: For the program

```
1  if (...) {
2    x = 2; y = 3;
3  } else {
4    x = 3; y = 2;
5  }
6  z = x + y;
7  ;
```

where each program location l_i corresponds to the line i before the line is executed, an MOP analysis enumerates this path set: $\{l_1\, l_2\, l_3\, l_6\, l_7,\ l_1\, l_4\, l_5\, l_6\, l_7\}$. If the properties that are computed are the constant values (constant propagation), the MOP analysis generates these properties for the first path:

$$\begin{array}{ll} L_{1a} = \{\} & L_{3a} = \{(\mathtt{x}, 2), (\mathtt{y}, 3)\} \\ L_{2a} = \{\} & L_{6a} = \{(\mathtt{x}, 2), (\mathtt{y}, 3)\} \\ & L_{7a} = \{(\mathtt{x}, 2), (\mathtt{y}, 3), (\mathtt{z}, 5)\} \end{array}$$

and these for the second path:

$$\begin{array}{ll} L_{1b} = \{\} & L_{5b} = \{(\mathtt{x}, 3), (\mathtt{y}, 2)\} \\ L_{4b} = \{\} & L_{6b} = \{(\mathtt{x}, 3), (\mathtt{y}, 2)\} \\ & L_{7b} = \{(\mathtt{x}, 3), (\mathtt{y}, 2), (\mathtt{z}, 5)\}. \end{array}$$

The sets are joined ($L_k = L_{ka} \sqcup L_{kb}$), giving the following annotations for the labels:

$$\begin{array}{ll} L_1 = \{\} & L_3 = \{(\mathtt{x}, 2), (\mathtt{y}, 3)\} \\ L_2 = \{\} & L_5 = \{(\mathtt{x}, 3), (\mathtt{y}, 2)\} \\ L_4 = \{\} & L_6 = \{(\mathtt{x}, \top), (\mathtt{y}, \top)\} \\ & L_7 = \{(\mathtt{x}, \top), (\mathtt{y}, \top), (\mathtt{z}, 5)\}. \end{array}$$

[3] More details on treating loops during the construction of program invariants are given in Sect. 16.6.

The analysis determines that z is constant 5 at l_7. In contrast, an MFP analysis calculates the properties directly on the program locations, like this:

$$\begin{array}{ll} L_1 = \{\} & L_3 = \{(\mathtt{x}, 2), (\mathtt{y}, 3)\} \\ L_2 = \{\} & L_5 = \{(\mathtt{x}, 3), (\mathtt{y}, 2)\} \\ L_4 = \{\} & L_6 = L_3 \sqcup L_5 = \{(\mathtt{x}, \top), (\mathtt{y}, \top)\} \\ & L_7 = \{(\mathtt{x}, \top), (\mathtt{y}, \top), (\mathtt{z}, \top)\}. \end{array}$$

In particular, the value of z at program location l_7 is calculated from the set L_6, and the transfer function for $\mathtt{z} = \mathtt{x} + \mathtt{y}$ computes $\top + \top = \top$. The example shows that an MOP analysis can be more precise than an MFP analysis, but the two results coincide if the transfer functions distribute over $\sqcup$ [77]. In practice, a hybrid approach is often taken, where the MFP algorithm is augmented by a limited program expansion and MOP computation, e.g., "property-oriented expansion" [22, 28, 115].

Extensions. The previous presentation used forward analysis, where input properties generate output properties, which are combined with union as the join operation. Iteration-until-convergence is a least-fixed-point calculation. It is possible to compute properties in a backward analysis (e.g., definitely-live-variables analysis), where intersection is the join operation; this is usually a greatest-fixed-point calculation (cf. Sect. 16.6.1).

Some analyses, e.g., partial redundancy elimination (PRE) [89] use both forward and backward analysis. PRE can be simplified into a two-step fixed-point computation, where the results of a backward analysis are complemented and used as input to a forward analysis [114]. This reformulation reveals several crucial insights:

1. A program's control-flow automaton can be defined as a Kripke structure, and expansion (unrolling) of the automaton is analogous to exploring the program's state space.
2. The value sets computed by a data-flow analysis can be represented as temporal-logic formulas, where the meaning of a formula is a value set.
3. The MFP algorithm operates like the algorithm for branching-time model checking, and the MOP algorithm operates like the algorithm for linear-time model checking.

These insights were documented earlier [81, 114, 115], and form the foundation for the remainder of this chapter.

16.2.3 Model Checking

Model checking enumerates the sequences of states that arise during a program's execution and decides whether the sequences of states satisfy a safety property. Other chapters in this Handbook develop several variants of the notions *program*, *property*, and *algorithm* for model checking, so here we merely compare and contrast model checking to data-flow analysis.

Program Representation. Rather than a syntax tree or a control-flow automaton, classic model checking operates on a directed graph whose nodes are the program's run-time states, connected by edges that define sequencing. The directed graph is typically infinite and must be represented by a recursively enumerable set of transition rules. There are three variants of the transition rules:

1. A *Kripke structure* (S, R, I) consists of a set S of states, a transition relation $R \subseteq S \times S$, and a map $I : S \to 2^{\Phi}$ that assigns to each state a subset of properties from property set Φ that hold for the state.
2. A *labeled transition system* $(S, Act, \to)$ consists of a set S of states, a set Act of actions (transfer functions), and a transition relation $\to \subseteq S \times Act \times S$.
3. A *Kripke transition system* $(S, Act, \to, I)$ combines the components of the two previous forms.

The details needed to represent even one state can be practically prohibitive, therefore states are often abstracted by forgetting details of the state's "content" or even by replacing a state by a set of propositions that hold true for the state—this is often done with the Kripke-structure representation.

At the other extreme, if the set S of states is defined as exactly the program locations, then a control-flow automaton is readily expressed [114] and program expansion is easily done [115]. The notion of *abstract reachability graphs* (ARGs) is often used in the context of software verification (cf. BLAST [13]).

Property Representation. Model checking uses a temporal logic as its property language—it is a logic because it includes conjunction, disjunction, and (usually) negation; it is temporal because it uses operators that are interpreted on the sequences of nodes in a path or graph. The properties might be

1. *path-based (linear time)*, e.g., $E\phi$ might mean "there exists a state along a path that validates proposition ϕ," or
2. *graph-based (branching time)*, e.g., $EF\psi$ might mean "there exists a path generated from the current state that includes a state that validates ψ."

More details about temporal logics are provided in Chap. 2. The two variants recall the property languages used for MOP- and MFP-based data-flow analyses. The connection stands out when one reconsiders classic definitions, like this one for MFP-based live-variable calculation [78]:

$$LiveVarsAt(l) = UsedAt(l) \cup \left(NotModifiedAt(l) \cap \left(\bigcup_{l' \in succ(l)} LiveVarsAt(l') \right) \right)$$

where l is a program location and $succ(l)$ is the set of all successor locations for location l in the control-flow automaton. The equation defines the set of possibly live variables at a program location l. Compare the definition to the following, coded in branching-time temporal logic, which holds for a program location when variable x is possibly live at that program location [107, 113]:

$$isLiveVar_{\mathrm{x}} = isUsed_{\mathrm{x}} \vee \left(\neg isModified_{\mathrm{x}} \wedge EF(isLiveVar_{\mathrm{x}}) \right).$$

We have that $x \in \mathit{LiveVarsAt}(l)$ iff $l \models \mathit{isLiveVar}_x$ for every program variable x [113]—*the temporal-logic formula defines the data-flow set*.

Analysis Algorithm. A model-checking algorithm answers queries, posed in temporal logic, about a program representation. The algorithm generates a graph or path set from the program representation (transition rules) and applies the interpretation function to the nodes in the graph (respectively, paths) to answer the query. There are numerous algorithms for performing this activity, but with the perspective provided in this chapter, we can say that a model-checking algorithm is an MFP (resp., MOP) calculation of the graph (resp., paths) generated from a program's transition rules for answering the branching-time (resp., linear-time) query.

The generated graph or paths might be infinite, thus answering queries is semi-decidable. A bound can be placed on the number of iterations or graph size ("bounded model checking") or a join operation ("widening" [38]) might be used to force the generated graph to be finite. (When the latter is used, the technique is sometimes called "abstract model checking.")

The remainder of this chapter develops several variations of *property* and *algorithm* that are inspired by the deep correspondence between data-flow analysis and model checking.

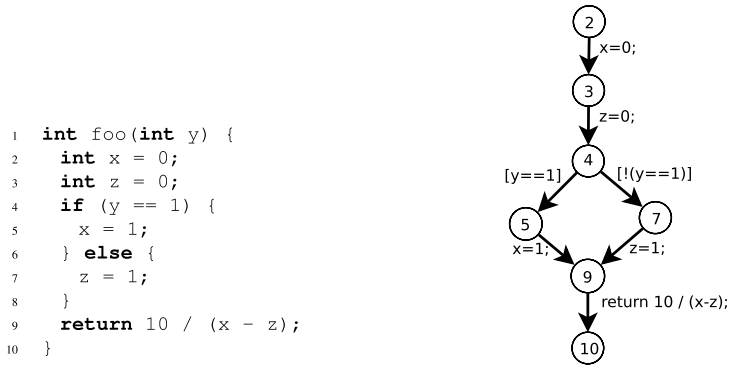
16.3 Unifying Formal Framework/Comparison of Algorithms

The previous section has outlined the differences, and similarities, between the three static-analysis techniques type checking, data-flow analysis, and model checking. The discussion was structured by the components of every static analysis: program representation, property representation, and analysis algorithm. In the following, we explain the unifying formal framework of *configurable program analysis*, which has successful implementations in software-verification tools (CPACHECKER [21], CPALIEN [91], CPATIGER [20], JAKSTAB [73]). The framework makes it possible to formalize each of the three techniques in the same formal setting.[4] In order to concretely explain the differences, we model the algorithms that were traditionally used for data-flow analysis and software model checking as instances of the framework.

16.3.1 Preliminaries

Control-Flow Automaton (CFA). A *program* is represented by a *control-flow automaton*. We restrict our formal presentation to simple imperative programming, where all operations are either assignments or assume operations (conditional ex-

[4] Other approaches have been proposed that address similar goals, for example, the fixed-point analysis machine [79, 80, 116].



(a) Example C function (b) Control-flow automaton (CFA)

Fig. 1 Example C function and corresponding CFA; the program locations in the CFA (**b**) correspond to the line numbers in the program text (**a**) before the line of code is executed

ecutions), all variables range over integers, and no function calls occur,[5] while we use C syntax to denote example program code. A CFA (L, l_0, G) consists of a set L of program locations (models the program counter pc), an initial program location l_0 (models the program entry), and a set $G \subseteq L \times Ops \times L$ of control-flow edges (models program operations that are executed when control flows from one program location to another). Program operations from Ops are either assignment operations or assume operations. The set of program variables that occur in program operations from Ops is denoted by X. A *concrete state* of a program is a variable assignment c that assigns a value to each variable from $X \cup \{pc\}$. The set of all concrete states of a program is denoted by C. A set $r \subseteq C$ of concrete states is called a *region*. Each edge $g \in G$ defines a (labeled) transition relation $\overset{g}{\rightarrow} \subseteq C \times \{g\} \times C$, which defines how concrete states of one program location (source) are transformed into concrete states of another program location (target). The complete transition relation $\rightarrow$ is the union over all control-flow edges: $\rightarrow = \bigcup_{g \in G} \overset{g}{\rightarrow}$. We write $c \overset{g}{\rightarrow} c'$ if $(c, g, c') \in \rightarrow$, and $c \rightarrow c'$ if there exists a g with $c \overset{g}{\rightarrow} c'$. A concrete state c_n is *reachable* from a region r, denoted by $c_n \in Reach(r)$, if there exists a sequence of concrete states $\langle c_0, c_1, \ldots, c_n \rangle$ such that $c_0 \in r$ and for all $1 \leq i \leq n$, we have $c_{i-1} \rightarrow c_i$.

Example 1 Figure 1 shows an example program (a) and the corresponding CFA (b). The CFA has seven program locations ($L = \{2, 3, 4, 5, 7, 9, 10\}, l_0 = 2$) and three program variables ($X = \{\mathrm{x}, \mathrm{y}, \mathrm{z}\}$). The initial region r_0 of this program is the set $\{c \in C \mid c(pc) = 2\}$. The only concrete state at program location 5 (i.e., before line 5 is executed) that is reachable from the initial region is the following variable assignment: $c(pc) = 5, c(\mathrm{x}) = 0, c(\mathrm{y}) = 1, c(\mathrm{z}) = 0$. The set of concrete states at program

[5] Tool implementations usually support interprocedural analysis, either via function inlining, function summaries, or other techniques [100, 116]. More information on this topic is given in Chap. 17.

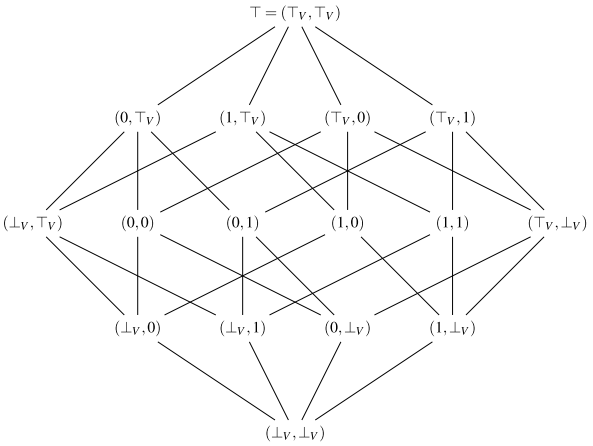


Fig. 2 Example lattice

location 9 that are reachable from the initial region can be represented by the predicate $pc = 9 \wedge ((\mathrm{x} = 1 \wedge \mathrm{y} = 1 \wedge \mathrm{z} = 0) \vee (\mathrm{x} = 0 \wedge \mathrm{y} \neq 1 \wedge \mathrm{z} = 1))$.

Semi-lattices. A *partial order* $\sqsubseteq \subseteq E \times E$ over a (possibly infinite) set E is a binary relation that is reflexive ($e \sqsubseteq e$ for all $e \in E$), transitive (if $e \sqsubseteq e'$ and $e' \sqsubseteq e''$ then $e \sqsubseteq e''$), and antisymmetric (if $e \sqsubseteq e'$ and $e' \sqsubseteq e$ then $e = e'$). The *least upper bound* for a subset $M \subseteq E$ of elements is the smallest element e such that $e' \sqsubseteq e$ for all $e' \in M$. The partial order $\sqsubseteq$ induces a *semi-lattice*[6] (defines the structure of the semi-lattice) if every subset $M \subseteq E$ has a least upper bound $e \in E$ (cf. [94] for more details). We denote a semi-lattice that is induced by a set E and a partial order $\sqsubseteq$ using the tuple $(E, \sqsubseteq, \sqcup, \top)$, in order to assign symbols to special components: the join operator $\sqcup : E \times E \to E$ yields the least upper bound for two elements (we use the set notation $\bigsqcup \{e_1, e_2, \ldots\}$ to denote $e_1 \sqcup e_2 \sqcup \ldots$) and the top element $\top$ is the least upper bound of the set E ($\top = \sqcup E$).

Example 2 Let us consider the semi-lattice $(V, \sqsubseteq, \sqcup, \top)$ that can be used for a constant-propagation analysis over two Boolean variables. The set V of lattice elements consists of variable assignments: $V = X \to \{\bot_V, 0, 1, \top_V\}$, $X = \{x_1, x_2\}$. The partial order $\sqsubseteq$ is defined as $v \sqsubseteq v'$ if $\forall x \in X : v(x) = v'(x)$ or $v(x) = \bot_V$ or $v'(x) = \top_V$. Figure 2 depicts this simple lattice as a graph. The nodes represent lattice elements, where a pair (c_1, c_2) denotes the variable assignment $\{x_1 \mapsto c_1, x_2 \mapsto c_2\}$. The edges represent the partial order (if read in the upwards direction), where reflexive and transitive edges are omitted. The top element $\top$ is the variable assignment with $\top(x) = \top_V$ for all $x \in X$.

[6] Sometimes, complete lattices are used in formalizations of data-flow analyses, but most practical analyses require only one operator: either the least upper bound or the greatest lower bound.

Program Analysis. A *program analysis* for a CFA (L, l_0, G) consists of an abstract domain D and a transfer relation $\rightsquigarrow$. The *abstract domain* $D = (C, \mathscr{E}, [\![\cdot]\!])$ is defined by the set C of concrete states, a semi-lattice $\mathscr{E} = (E, \sqsubseteq, \sqcup, \top)$, and a concretization function $[\![\cdot]\!]$. The lattice elements are used as (components of) abstract states in the program analysis. Each abstract state represents a (possibly infinite) set of concrete states. The concretization function $[\![\cdot]\!] : E \to 2^C$ assigns to each abstract state its meaning, i.e., the set of concrete states that it represents. The abstract domain determines the objective of the analysis, i.e., the aspects of the program that are analyzed. The *transfer relation* $\rightsquigarrow \subseteq E \times G \times E$ assigns to each abstract state e possible new abstract states e' that are abstract successors of e, and each transfer is labeled with a control-flow edge g. We write $e \overset{g}{\rightsquigarrow} e'$ if $(e, g, e') \in \rightsquigarrow$, and $e \rightsquigarrow e'$ if there exists a g with $e \overset{g}{\rightsquigarrow} e'$. A program analysis has to fulfill certain requirements for soundness, i.e., to guarantee that no violations of the property are missed by the analysis [17, 38, 94].

Example 3 Considering the example in Fig. 2 again, the concretization function $[\![\cdot]\!]$ relates the lattice elements to sets of variable assignments. For example, lattice element $(1, 0)$ maps the first variable to value 1 and the second variable to value 0. The lattice element $\top$ represents all concrete states. Given a variable x, we use the bottom element $\bot_V$ to denote the variable assignment that assigns no value to variable x (representing the empty set of concrete states). Note that in a program analysis, there might be several (strictly speaking different) lattice elements that represent the empty set of concrete states: every variable assignment that has (at least) one variable assigned to $\bot_V$ cannot represent any concrete state.[7]

16.3.2 Algorithm of Data-Flow Analysis

We now present an iteration algorithm for MFP data-flow analysis. According to classic definitions of data-flow analysis [94], the algorithm computes, for a given abstract domain, a function reached that assigns to each analyzed program location an abstract data state (i.e., the abstract states consist of a program location and an abstract data state, the latter represented by a lattice element).

Algorithm 1(a) operates on a partial function and a set: the function reached represents the result of the data-flow analysis, i.e., the mapping from program locations to abstract data states; the set waitlist represents the program locations for which the abstract data state was changed, i.e., the fixed point is not reached as long as waitlist is not empty. Algorithm 1(a) is guaranteed to terminate if the semi-lattice has finite height; the run time depends on the height of the semi-lattice and the number of program locations. The algorithm starts by assigning the initial abstract data state

[7] This leads to the notion of "smashed bottom," where all variable assignments with at least one variable assigned to $\bot_V$ are subsumed by one representative ($\bot$). We do not emphasize this notion in our chapter.

Algorithm 1 Typical differences of data-flow analysis (Algorithm *DFA*) and software model checking (Algorithm *Reach*)

Input: set L of locations, an abstract domain E, transfer relation $\rightsquigarrow$, initial abstract state (l_0, e_0) with $l_0 \in L, e_0 \in E$
Output: set of reachable abstract states (pairs of location and abstract data state)

(a) Algorithm *DFA*$(L, E, \rightsquigarrow, e_0)$

Variables: function reached : $L \rightharpoonup E$, set waitlist $\subseteq L$

```
 1: waitlist := {l0}
 2: reached(l0) := e0
 3: while waitlist ≠ {} do
 4:    choose l from waitlist
 5:    waitlist := waitlist \ {l}
 6:    for each (l', e') with (l, e)⇝(l', e') do
 7:       // if not already covered
 8:       if e' ⋢ reached(l') then
 9:          // join with existing abstract data state
10:          reached(l') := reached(l') ⊔ e'
11:          waitlist := waitlist ∪ {l'}
12: return reached
```

(b) Algorithm *Reach*$(L, E, \rightsquigarrow, e_0)$

Variables: set reached $\subseteq L \times E$, set waitlist $\subseteq L \times E$

```
 1: waitlist := {(l0, e0)}
 2: reached := {(l0, e0)}
 3: while waitlist ≠ {} do
 4:    choose (l, e) from waitlist
 5:    waitlist := waitlist \ {(l, e)}
 6:    for each (l', e') with (l, e)⇝(l', e') do
 7:       // if not already covered
 8:       if ∄(l', e'') ∈ reached : e' ⊑ e'' then
 9:          // add as new abstract state
10:          reached := reached ∪ {(l', e')}
11:          waitlist := waitlist ∪ {(l', e')}
12: return reached
```

to the initial program location. Then it iterates through the `while` loop until the set waitlist is empty. In every loop iteration, one program location is taken out of the waitlist and abstract successors are computed for the corresponding successor program locations. The abstract data element for the successor program location in function reached is added for the program location, or the old abstract data state is replaced by the join of the old and new abstract data states. Because we operate on a partial function reached, we extend $e' \not\sqsubseteq \mathsf{reached}(l')$ to return *false* if $\mathsf{reached}(l')$ is undefined, and we extend $\mathsf{reached}(l') \sqcup e'$ to $\bigsqcup(\{e'' \mid (l', e'') \in \mathsf{reached}\} \cup e')$.[8]

Example 4 Consider the example program from Fig. 1 and an abstract domain for constant propagation; suppose the verification task is to ensure that no division by zero occurs. The data-flow analysis computes a function reached with the following entries: $2 \mapsto \{\mathtt{x} = \top, \mathtt{y} = \top, \mathtt{z} = \top\}$, $3 \mapsto \{\mathtt{x} = 0, \mathtt{y} = \top, \mathtt{z} = \top\}$, and $4 \mapsto \{\mathtt{x} = 0, \mathtt{y} = \top, \mathtt{z} = 0\}$. Following the `then` branch from program location 4, the algorithm computes the entries $5 \mapsto \{\mathtt{x} = 0, \mathtt{y} = 1, \mathtt{z} = 0\}$ and $9 \mapsto \{\mathtt{x} = 1, \mathtt{y} = 1, \mathtt{z} = 0\}$, and stores them in the function reached. For the `else` branch, the algorithm computes the entries $7 \mapsto \{\mathtt{x} = 0, \mathtt{y} = \top, \mathtt{z} = 0\}$ and $9 \mapsto \{\mathtt{x} = 0, \mathtt{y} = \top, \mathtt{z} = 1\}$. Since reached already has an entry for program location 9, the two abstract data states are joined, which results in the entry $9 \mapsto \{\mathtt{x} = \top, \mathtt{y} = \top, \mathtt{z} = \top\}$. The correctness of the program (in terms of division by zero) cannot be established.

[8] Alternative formalizations use total functions for reached and require some lower bound $\bot$ to exist in the (semi-) lattice, which is used as the initial abstract state to make reached total.

16.3.3 Algorithm of Model Checking

We now consider an iteration algorithm for software model checking. According to the classic reachability algorithm, the algorithm computes the nodes of an abstract reachability tree [13], which contains all reachable abstract states according to the transfer relation. In difference to the data-flow analysis, the join operation is never applied.

Algorithm 1(b) operates on two sets reached and waitlist, which are initialized with a pair of the initial control-flow location and the initial abstract data state. In every iteration of the `while` loop, the algorithm takes one abstract state from the set waitlist and computes successors, as long as the fixed point is not reached. Algorithm 1(b) is not guaranteed to terminate if the semi-lattice is infinite; software model checking in general is a semi-decidable analysis. If there is no abstract state in the set reached that entails the new abstract state, then the new abstract state is added to the sets reached and waitlist. The join operation is never called, and thus, the set of reached abstract states contains all nodes that an abstract reachability tree (ART) [13] would contain (the edges of the actual tree are not necessarily stored; but many model-checking algorithms do store an ART to support certain features, such as error-path analysis [34]).

Example 5 We reconsider the example program from Fig. 1 and an abstract domain for constant propagation. The model-checking algorithm computes a set reached with the following entries: $(2, \{\mathtt{x} = \top, \mathtt{y} = \top, \mathtt{z} = \top\})$, $(3, \{\mathtt{x} = 0, \mathtt{y} = \top, \mathtt{z} = \top\})$, and $(4, \{\mathtt{x} = 0, \mathtt{y} = \top, \mathtt{z} = 0\})$. Following the `then` branch from program location 4, the algorithm computes the entries $(5, \{\mathtt{x} = 0, \mathtt{y} = 1, \mathtt{z} = 0\})$ and $(9, \{\mathtt{x} = 1, \mathtt{y} = 1, \mathtt{z} = 0\})$, and stores them in the set reached. For the `else` branch, the algorithm computes the entries $(7, \{\mathtt{x} = 0, \mathtt{y} = \top, \mathtt{z} = 0\})$ and $(9, \{\mathtt{x} = 0, \mathtt{y} = \top, \mathtt{z} = 1\})$. Although reached already has an entry for program location 9, this second entry is stored in the set reached, and the correctness of the example program (in terms of division by zero) is established: the value of variable `x` is always different from the value of variable `z`.

16.3.4 Unified Algorithm Using Configurable Program Analysis

In theory, data-flow analysis and model checking have the same expressive power [108]. In this section, we explain the unifying framework of configurable program analysis [17, 18], a formalism and algorithm that makes it possible to *practically* unify the approaches. Comparing the two Algorithms 1(a) and (b) again reveals the similarity that motivates a unified algorithm, and also the differences that motivate the configurable operators merge and stop, which we will define below as part of the configurable program analysis and then use in the unified Algorithm 2.

Configurable Program Analysis (CPA). A *configurable program analysis* $\mathbb{D} = (D, \rightsquigarrow, \mathsf{merge}, \mathsf{stop})$ for a CFA (L, l_0, G) consists of an abstract domain D, a transfer relation $\rightsquigarrow$, a merge operator merge, and a termination check stop, which are explained in the following. These four components *configure* our algorithm and influence the precision and cost of a program analysis.

1. The *abstract domain* $D = (C, \mathscr{E}, [\![\cdot]\!])$ is defined by the set C of concrete states, a semi-lattice $\mathscr{E} = (E, \sqsubseteq, \sqcup, \top)$, and a concretization function $[\![\cdot]\!]$.
2. The *transfer relation* $\rightsquigarrow \subseteq E \times G \times E$ assigns to each abstract state e possible new abstract states e' that are abstract successors of e, and each transfer is labeled with a control-flow edge g.
3. The *merge operator* $\mathsf{merge} : E \times E \to E$ combines the information of two abstract states. The operator *weakens* the abstract state (also called *widening*) that is given as second parameter depending on the first parameter (the result of $\mathsf{merge}(e, e')$ can be anything between e' and $\top$).

   Note that the operator merge is not commutative, and is not necessarily the same as the join operator $\sqcup$ of the lattice, but merge can be based on $\sqcup$. Later we will use the following merge operators: $\mathsf{merge}^{sep}(e, e') = e'$ and $\mathsf{merge}^{join}(e, e') = e \sqcup e'$.
4. The *termination check* $\mathsf{stop} : E \times 2^E \to \mathbb{B}$ checks whether the abstract state e that is given as first parameter is covered by the set R of abstract states given as second parameter, i.e., every concrete state that e represents is represented by some abstract state from R. The termination check can, for example, go through the elements of the set R that is given as second parameter and search for a single element that subsumes ($\sqsubseteq$) the first parameter, or—if D is a power-set domain[9]—can join the elements of R to check whether $\bigsqcup R$ subsumes the first parameter.

   Note that the termination check stop is not the same as the partial order $\sqsubseteq$ of the lattice, but stop can be based on $\sqsubseteq$. Later we will use the following termination checks (the second requires a power-set domain): $\mathsf{stop}^{sep}(e, R) = (\exists e' \in R : e \sqsubseteq e')$ and $\mathsf{stop}^{join}(e, R) = (e \sqsubseteq \bigsqcup R)$.

The abstract domain on its own does not determine the precision of the analysis; each of the four configurable components (abstract domain, transfer relation, merge operator, and termination check) independently contribute to adjusting both precision and cost of the analysis.

Unified Algorithm. In order to experiment with both data-flow analysis and model checking in one single algorithm, we unify the two algorithms using the operators merge and stop of the configurable program analysis.

Algorithm 2 abstracts from program locations and operates on two sets of abstract states (abstract-domain elements), i.e., the program location is represented in the abstract domain and is not specially treated anymore (classic data-flow analyses

[9] A *power-set domain* is an abstract domain such that $[\![e_1 \sqcup e_2]\!] = [\![e_1]\!] \cup [\![e_2]\!]$.

Algorithm 2 *CPA*($\mathbb{D}$, e_0)

Input: a CPA $\mathbb{D} = (D, \rightsquigarrow, \mathsf{merge}, \mathsf{stop})$,
an initial abstract state $e_0 \in E$, where E denotes the set of elements of the lattice of D
Output: a set of reachable abstract states
Variables: a set reached $\subseteq E$, a set waitlist $\subseteq E$

```
 1: waitlist := {e_0}
 2: reached := {e_0}
 3: while waitlist ≠ {} do
 4:    choose e from waitlist
 5:    waitlist := waitlist \ {e}
 6:    for each e' with e ⇝ e' do
 7:       for each e'' ∈ reached do
 8:          // combine with existing abstract state
 9:          e_new := merge(e', e'')
10:          if e_new ≠ e'' then
11:             waitlist := (waitlist ∪ {e_new}) \ {e''}
12:             reached := (reached ∪ {e_new}) \ {e''}
13:       if ¬ stop(e', reached) then
14:          waitlist := waitlist ∪ {e'}
15:          reached := reached ∪ {e'}
16: return reached
```

rely on an explicit representation of the program location). The sets reached and waitlist are initialized with the initial abstract state for the given configurable program analysis. As in the previous algorithms, every iteration of the loop processes one element from the set waitlist, and computes all abstract successors for that abstract state. The set waitlist is empty if the fixed point of the iteration is reached.

Now, for every abstract successor state, the algorithm merges the new abstract state with every existing abstract state in the set reached. It depends solely on the merge operator how often abstract states from reached are combined and how abstractly they are combined. In the case that the merge operator does not produce a new combined state, it simply returns the existing abstract state that was given as second parameter. Otherwise, it returns a new abstract state that entails the existing abstract state. In the latter case, the existing abstract state is removed from the sets reached and waitlist and the new abstract state is added to the sets reached and waitlist. (Obviously, an efficient implementation of the algorithm applies optimization to the `for each` loop from line 7 to line 12, e.g., using partitions or projections for the set reached.)

After the current abstract successor state has been merged with all existing abstract states, the stop operator determines whether the algorithm needs to store the current abstract state in the sets reached and waitlist. For example, if all concrete states that are represented by the current abstract state are covered (i.e., also represented) by existing abstract states in the set reached, then the current state may be ignored.

The set reached, at the fixed point of the iteration, represents the program invariant. Such fixed-point iterations and several other algorithms for constructing program invariants are discussed in Sect. 16.6.

16.3.5 Discussion

Effectiveness. The *effectiveness* of an analysis refers to the degree of precision with which the analysis determines whether a program satisfies or violates a given specification (number of false positives and false negatives). Model checking has a high degree of precision, due to the fact that all reachable abstract states are stored separately in the set of reachable states, i.e., model checking is automatically path-sensitive due to never applying join operations (if the set of reachable abstract states is seen as the reachability tree that represents execution paths). Data-flow analysis is often imprecise, when join operations are applied in order to reduce two abstract states to one. In comparison to standard data-flow analysis, power-set constructions for increasing the precision (e.g., for making an analysis path-sensitive) are not necessary in a configurable program analysis: the effect can easily be achieved by setting the merge operator to not join.

This is the strength of defining program analyses as CPA: the components abstract domain, transfer relation, merge operator, and stop operator separate concerns and provide a flexible way of tuning these components or exchanging them with others. For example, the merge operator encodes whether the algorithm works like MFP, or MOP, or uses a hybrid approach (cf. the merge operator used in adjustable-block encoding [22]). Each of the components has an important impact on the precision and performance of the program analysis.

Efficiency. The *efficiency* (also called performance) of an analysis measures the resource consumption of an algorithm (in time or space). The resources required for an analysis often decide whether the analysis should be applied to the problem or not. For example, the run time of a data-flow analysis is determined by the height of the abstract domain's lattice, the size of the control-flow automaton, and the number of variables in the program. Most of the classic data-flow analyses are efficient (low polynomial run time) and can be used in compilers for optimization. Model checking sometimes requires resources exponential in the program size (if terminating at all). Due to the high precision of typical model-checking domains, such as predicate abstraction, the sub-problem of computing an abstract successor state is often NP-hard already.

Iteration Order. The *iteration order* defines the sequence in which abstract states from the set waitlist are processed by the exploration algorithm. We did not discuss this parameter because it is orthogonal to the difference between data-flow analysis and model checking, i.e., most iteration orders can be used for both techniques. In Algorithm 2, the iteration order is implemented in the operator choose. The most simple iteration orders are breadth-first search (BFS) and depth-first search (DFS). The iteration order DFS is often not advisable for data-flow analysis, because after each join operation, the algorithm has to re-explore all successors of abstract states that represent more concrete states after the join. For model checking, both orders are applicable, while some existing implementations of model-checking tools prefer the DFS order (e.g., BLAST [13]). The best iteration order is often a combination of

both, for example by using a topological (reverse post-order) algorithm in which DFS is performed until a meet point, while further exploration has to wait for the control flow to arrive via all other branches [22]. Also chaotic iteration orders [29] were investigated and found to be useful. More details can be found in Sect. 16.6.1.

16.3.6 Composition of Configurable Program Analyses

Different CPAs have different strengths and weaknesses, and therefore, we need to construct combinations of component analyses to pick the advantages of several components, in order to achieve more effective program analyses.

Composite. A configurable program analysis can be composed of several configurable program analyses [17]. A *composite program analysis* $\mathscr{C} = (\mathbb{D}_1, \mathbb{D}_2, \rightsquigarrow_\times, \mathsf{merge}_\times, \mathsf{stop}_\times)$[10] consists of two configurable program analyses $\mathbb{D}_1$ and $\mathbb{D}_2$ sharing the same set C of concrete states with E_1 and E_2 being their respective sets of abstract states, a composite transfer relation $\rightsquigarrow_\times \subseteq (E_1 \times E_2) \times G \times (E_1 \times E_2)$, a composite merge operator $\mathsf{merge}_\times : (E_1 \times E_2) \times (E_1 \times E_2) \to (E_1 \times E_2)$, and a composite termination check $\mathsf{stop}_\times : (E_1 \times E_2) \times 2^{E_1 \times E_2} \to \mathbb{B}$. The three composites $\rightsquigarrow_\times$, $\mathsf{merge}_\times$, and $\mathsf{stop}_\times$ are expressions over the components of $\mathbb{D}_1$ and $\mathbb{D}_2$ ($\rightsquigarrow_i$, merge_i, stop_i, $[\![\cdot]\!]_i$, E_i, $\sqsubseteq_i$, $\sqcup_i$, $\top_i$), as well as the operators $\downarrow$ and $\preceq$ (defined below). The composite operators can manipulate lattice elements only through those components, never directly (e.g., if $\mathbb{D}_1$ is already a result of a composition, then we cannot access the tuple elements of abstract states from E_1, nor redefine merge_1). The only way of using additional information is through the operators $\downarrow$ and $\preceq$.

Strengthen. The *strengthening* operator $\downarrow : E_1 \times E_2 \to E_1$ computes a stronger element from the lattice set E_1 by using the information of a lattice element from E_2; it has to meet the requirement $\downarrow(e, e') \sqsubseteq e$. The strengthening operator can be used to define a composite transfer relation $\rightsquigarrow_\times$ that is stronger than a direct product relation. For example, if we combine predicate analysis and constant propagation, the strengthening operator $\downarrow_{\mathbb{CO},\mathbb{P}}$ can "sharpen" the explicit-value assignment of the constant propagation (cf. Sect. 16.4.2) by considering the predicates in the predicate analysis (cf. Sect. 16.4.4).

Compare. Furthermore, we allow the definitions of composite operators to use the *compare* relation $\preceq \subseteq E_1 \times E_2$, to compare elements of different lattices.

Composition. For a given composite program analysis $\mathscr{C} = (\mathbb{D}_1, \mathbb{D}_2, \rightsquigarrow_\times, \mathsf{merge}_\times, \mathsf{stop}_\times)$, we can construct the configurable program analysis $\mathbb{D}_\times = (D_\times, \rightsquigarrow_\times, \mathsf{merge}_\times, \mathsf{stop}_\times)$, where the product domain $D_\times$ is defined as the direct product

[10] We extend this notation to any finite number of $\mathbb{D}_i$.

of D_1 and D_2: $D_\times = D_1 \times D_2 = (C, \mathscr{E}_\times, [\![\cdot]\!]_\times)$. The product lattice is $\mathscr{E}_\times = \mathscr{E}_1 \times \mathscr{E}_2 = (E_1 \times E_2, \sqsubseteq_\times, \sqcup_\times, (\top_1, \top_2))$ with $(e_1, e_2) \sqsubseteq_\times (e'_1, e'_2)$ if $e_1 \sqsubseteq_1 e'_1$ and $e_2 \sqsubseteq_2 e'_2$ (and for the join operation the following holds $(e_1, e_2) \sqcup_\times (e'_1, e'_2) = (e_1 \sqcup_1 e'_1, e_2 \sqcup_2 e'_2)$). The product concretization function $[\![\cdot]\!]_\times$ is such that $[\![(d_1, d_2)]\!]_\times = [\![d_1]\!]_1 \cap [\![d_2]\!]_2$.

The literature agrees that this direct product itself is often not sharp enough [36, 39]. Even improvements over the direct product (e.g., the reduced product [28, 39] or the logical product [61]) do not solve the problem completely. However, in a configurable program analysis, we can specify the desired degree of "sharpness" in the composite operators $\rightsquigarrow_\times$, $\mathsf{merge}_\times$, and $\mathsf{stop}_\times$. For a given product domain, the definitions of the three composite operators determine the precision of the resulting configurable program analysis. In previous approaches, a redefinition of basic operations was necessary, but using configurable program analysis, we can reuse the existing abstract interpreters. For certain numerical abstract domains, the composite transfer relation can be automatically constructed: if the abstract domains of two given CPAs fulfill certain requirements (convex, stably infinite, disjoint) then the most precise abstract transfer relation can be computed [61].

16.4 Classic Examples (Component Analyses)

We now define and explain some well-known classic example analyses, in order to demonstrate the formalism of configurable program analysis. We use the notations that were introduced in Sects. 16.3.1, 16.3.4, and 16.3.6.

16.4.1 Reachable-Code Analysis

The reachable-code analysis (also known as dead-code analysis) identifies all locations of the control-flow automaton that can be reached from the program entry location. This classic analysis tracks only syntactic reachability, i.e., the operations are not interpreted.

The *location analysis* is a configurable program analysis $\mathbb{L} = (D_\mathbb{L}, \rightsquigarrow_\mathbb{L}, \mathsf{merge}_\mathbb{L}, \mathsf{stop}_\mathbb{L})$ that tracks the reachability of program locations and consists of the following components:

1. The abstract domain $D_\mathbb{L}$ is based on the semi-lattice for the set L of program locations: $D_\mathbb{L} = (C, \mathscr{L}, [\![\cdot]\!])$, with $\mathscr{L} = (L \cup \{\top\}, \sqsubseteq, \sqcup, \top)$ (also called a "flat semi-lattice"), $l \sqsubseteq \top$, and $l \neq l' \Rightarrow l \not\sqsubseteq l'$ for all elements $l, l' \in L$ (this implies $\top \sqcup l = \top, l \sqcup l' = \top$ for all elements $l, l' \in L, l \neq l'$), and $[\![\top]\!] = C$, and for all $l \in L$: $[\![l]\!] = \{c \in C \mid c(pc) = l\}$.

   The element $\top$ represents the fact that the program location is not known.
2. The transfer relation $\rightsquigarrow_\mathbb{L}$ has the transfer $l \overset{g}{\rightsquigarrow}_\mathbb{L} l'$ if $g = (l, op, l')$, and has the transfer $\top \overset{g}{\rightsquigarrow}_\mathbb{L} \top$ for all $g \in G$.

The transfer relation determines the syntactic successor in the CFA without considering the semantics of the operation *op*.

3. The merge operator does not combine elements when the control flow meets: $\mathsf{merge}_{\mathbb{L}} = \mathsf{merge}^{sep}$.
4. The termination check considers abstract states individually: $\mathsf{stop}_{\mathbb{L}} = \mathsf{stop}^{sep}$.

This (simple) abstract domain can be used to perform a syntactic reachability analysis, for example to eliminate control-flow operations that can never be executed. More importantly, this CPA can be used to track the program location when combined with other CPAs, in order to separate the concern of location tracking from other analyses. In practice, a semantic reachable-code analysis would be preferred to search for dead code, for example using a predicate analysis, as was done in the context of model-checking-based test-case generation [7]. More details about the connection between model checking and testing are provided in Chap. 19.

16.4.2 Constant Propagation

The constant-propagation analysis identifies variables that store constant values at certain program locations, i.e., at a given program location, the value is always the same. This classic domain of data-flow analysis can be used to reduce the number of variables in a program by substituting constants for variables.

The *constant-propagation analysis* is a configurable program analysis $\mathbb{CO} = (D_{\mathbb{CO}}, \rightsquigarrow_{\mathbb{CO}}, \mathsf{merge}_{\mathbb{CO}}, \mathsf{stop}_{\mathbb{CO}})$ that tries to determine, for each program location, the value of each variable, and consists of the following components (we use the set L of program locations, the set $X \neq \{\}$ of program variables, and the set $\mathbb{Z}$ of integer values):

1. The abstract domain $D_{\mathbb{CO}} = (C, \mathscr{E}, [\![\cdot]\!])$ consists of the following three components. The set C is the set of concrete states. The semi-lattice $\mathscr{E}$ represents the abstract states, which store for a program location an abstract variable assignment. Formally, the semi-lattice $\mathscr{E} = ((L \cup \{\top_L\}) \times (X \to \mathscr{Z}), \sqsubseteq, \sqcup, (\top_L, v_\top))$, with $\mathscr{Z} = \mathbb{Z} \cup \{\top_{\mathscr{Z}}\}$, is induced by the partial order $\sqsubseteq$ that is defined as $(l, v) \sqsubseteq (l', v')$ if $(l = l' \text{ or } l' = \top_L)$ and $\forall x \in X : v(x) = v'(x)$ or $v'(x) = \top_{\mathscr{Z}}$. (The join operator $\sqcup$ yields the least upper bound, and $v_\top$ is the abstract variable assignment with $v_\top(x) = \top_{\mathscr{Z}}$ for each $x \in X$.) A concrete state c matches a program location l if $c(pc) = l$ or $l = \top_L$. Similarly, a concrete state c is compatible with an abstract variable assignment v if for all $x \in X$, $c(x) = v(x)$ or $v(x) = \top_{\mathscr{Z}}$. The concretization function $[\![\cdot]\!]$ assigns to an abstract state (l, v) all concrete states that match the program location l and are compatible with the abstract variable assignment v.
2. The transfer relation $\rightsquigarrow_{\mathbb{CO}}$ has the transfer $(l, v) \overset{g}{\rightsquigarrow} (l', v')$ if

   (1) $g = (l, \texttt{assume}(p), l')$ and $\phi(p, v)$ is satisfiable and for all $x \in X$:

$$v'(x) = \begin{cases} c & \text{if } c \text{ is the only satisfying assignment of } \phi(p, v) \\ & \text{for variable } x \\ v(x) & \text{otherwise} \end{cases}$$

where, given a predicate p over variables in X and an abstract variable assignment v, we define $\phi(p, v) := p \wedge \bigwedge_{x \in X, v(x) \neq \top_{\mathscr{Z}}} x = v(x)$ or

(2) $g = (l, \mathtt{w} := e, l')$ and for all $x \in X$:

$$v'(x) = \begin{cases} eval(e, v) & \text{if } x = \mathtt{w} \\ v(x) & \text{otherwise} \end{cases}$$

where, given an expression e over variables in X and an abstract variable assignment v,

$$eval(e, v) := \begin{cases} \top_{\mathscr{Z}} & \text{if } v(x) = \top_{\mathscr{Z}} \text{ for some } x \in X \text{ that occurs in } e \\ z & \text{otherwise, where expression } e \text{ evaluates to } z \text{ when} \\ & \text{each variable } x \text{ is replaced by } v(x) \text{ in } e \end{cases}$$

or

(3) $l = l' = \top_L$ and $v' = v_\top$.

3. The merge operator is defined by

$$\mathsf{merge}_{\mathbb{CO}}\big((l, v), (l', v')\big) = \begin{cases} (l, v) \sqcup (l', v') & \text{if } l = l' \\ (l', v') & \text{otherwise} \end{cases}$$

(The two abstract variable assignments are combined where the control flow meets.)

4. The termination check is defined by $\mathsf{stop}_{\mathbb{CO}} = \mathsf{stop}^{sep}$.

Example 6 Consider the C function in Fig. 1(a) again, and construct a CPA for constant propagation. The following lattice element is an example of an abstract state that is reachable in the program code from Fig. 1(a): $(4, \{x \mapsto 0, y \mapsto \top_{\mathscr{Z}}, z \mapsto 0\})$.

Note that CPA $\mathbb{CO}$ performs an MFP computation, which is not precise enough for proving the correctness of the function in Fig. 1(a). If we change the merge operator $\mathsf{merge}_{\mathbb{CO}}$ to merge^{sep}, then we move from MFP to what corresponds to abstract reachability trees (never join). This changed analysis is similar to explicit-value analysis [24]. Explicit-state model checking is discussed in Chap. 5.

Example 7 Considering the example from Fig. 1 again, but using merge^{sep} as merge operator, we obtain the following two different abstract states for program location 9: $(9, \{\mathtt{x} \mapsto 1, \mathtt{y} \mapsto 1, \mathtt{z} \mapsto 0\})$ and $(9, \{\mathtt{x} \mapsto 0, \mathtt{y} \mapsto \top_{\mathscr{Z}}, \mathtt{z} \mapsto 1\})$. This proves that a division by zero is not possible.

16.4.3 Reaching Definitions

The reaching-definitions analysis computes for every program location and for every variable a set of assignment operations that may have defined the value of the

variable (i.e., definitions that "reach" the location). This classic domain of data-flow analysis (very similar to use-def analysis) is used in compiler optimization to infer dependencies between operations [2], and in code-structure analysis to quantitatively measure the data-flow [12]. Furthermore, in selective test-case generation [102], an efficient use-def analysis is necessary to determine which program locations need to be covered by a test case (for a given changed definition, we need to compute all uses of that definition).

An assigning CFA edge e is a *reaching definition* for program location l and variable x if there exists a path in the CFA through edge e to program location l without any (re-)definition of x (compare with Sect. 16.2.3).

The *reaching-definitions analysis* is a configurable program analysis $\mathbb{RD} = (D_{\mathbb{RD}}, \rightsquigarrow_{\mathbb{RD}}, \mathsf{merge}_{\mathbb{RD}}, \mathsf{stop}_{\mathbb{RD}})$, which computes the set of reaching definitions for each program location, and consists of the following components (X is the set of program variables):

1. The abstract domain $D_{\mathbb{RD}} = (C, \mathscr{E}, [\![\cdot]\!])$ consists of the set C of concrete states, the semi-lattice $\mathscr{E}$, and the concretization function $[\![\cdot]\!]$. The semi-lattice is given by $\mathscr{E} = ((L \cup \{\top_L\}) \times 2^E, \sqsubseteq_{\mathbb{RD}}, \sqcup_{\mathbb{RD}}, (\top_L, E))$, where $E \subseteq X \times (L \times L)$ is the set of definitions (variables paired with their defining edge) in the program, and we define $(l, S) \sqsubseteq_{\mathbb{RD}} (l', S')$ if $(l = l' \text{ or } l' = \top_L)$ and $S \subseteq S'$, which implies the join operator:

$$(l, S) \sqcup_{\mathbb{RD}} (l', S') = \begin{cases} (l, S \cup S') & \text{if } l = l' \\ (\top_L, S \cup S') & \text{otherwise.} \end{cases}$$

2. The transfer relation $\rightsquigarrow_{\mathbb{RD}}$ has the transfer $(l, S) \rightsquigarrow (l', S')$ if

   (1) there exists a CFA edge $g = (l, op, l') \in G$ and

$$S' = \begin{cases} (S \setminus \{(\mathtt{x}, k, k') \mid k, k' \in L\}) \cup \{(\mathtt{x}, l, l')\} & \text{if } op \text{ has the form } \mathtt{x := <expr>;} \\ S & \text{otherwise} \end{cases}$$

   or

   (2) $l = l' = \top_L$ and $S' = E$.

3. The merge operator is defined as

$$\mathsf{merge}_{\mathbb{RD}}((l, S), (l', S')) = \begin{cases} (l', S \cup S') & \text{if } l = l' \\ (l', S') & \text{otherwise.} \end{cases}$$

   (The two sets of reaching definitions are united where the control flow meets.)
4. The termination check is defined as $\mathsf{stop}_{\mathbb{RD}} = \mathsf{stop}^{sep}$.

Example 8 In the program of Fig. 1, variable x has the following reaching definitions at location 9: $\{(x, 2, 3), (x, 5, 9)\}$.

16.4.4 Predicate Analysis

For a given formula ϕ and a set π of predicates, the *Cartesian predicate abstraction* $(\phi)^{\pi}_{C}$ is the strongest conjunction of predicates from π that is implied by ϕ, and the *Boolean predicate abstraction* $(\phi)^{\pi}_{B}$ is the strongest Boolean combination of predicates from π that is implied by ϕ.

Predicate analysis is a program analysis that uses predicate abstraction to construct abstract states. The precision π of the predicate analysis is a finite set of predicates that controls the coarseness of the over-approximation of the abstract states. The precision can be refined during the analysis using CEGAR [34] and interpolation [69], and there can be different values for the precision at different program locations using lazy abstraction refinement [13, 71], however, for simplicity of presentation, we assume a fixed set of predicates. This classic domain of software model checking became popular and successful in the last decade due to the recent breakthroughs in decision procedures (SMT solvers) for Boolean formulas over expressions in the theory of linear arithmetic (LA) and equality with uninterpreted functions (EUF).

The Cartesian *predicate analysis* is a configurable program analysis $\mathbb{P} = (D_{\mathbb{P}}, \rightsquigarrow_{\mathbb{P}}, \mathsf{merge}_{\mathbb{P}}, \mathsf{stop}_{\mathbb{P}})$, which uses Cartesian predicate abstraction and consists of the following components (where the precision is given by the finite set π of predicates over the set X of program variables, with $\mathit{false} \in \pi$, that are tracked by the analysis; for a set $r \subseteq \pi$ of predicates, we write φ_r to denote the conjunction of all predicates in r, in particular $\varphi_{\{\}} = \mathit{true}$):

1. The domain $D_{\mathbb{P}} = (C, \mathscr{P}, [\![\cdot]\!])$ is based on the idea that regions are represented by conjunctions over a finite set of predicates. The semi-lattice is given as $\mathscr{P} = (2^{\pi}, \sqsubseteq, \sqcup, \top)$, where the partial order $\sqsubseteq$ is defined as $r \sqsubseteq r'$ if $r \supseteq r'$ (note that if $r \sqsubseteq r'$ then φ_r implies $\varphi_{r'}$). The least upper bound $r \sqcup r'$ is given by $r \cap r'$ (note that $\varphi_{r \sqcup r'}$ is implied by $\varphi_r \vee \varphi_{r'}$). The element $\top = \{\}$ leaves the abstract state unconstrained (*true*), i.e., every concrete state is represented. We used the subsets of π as the lattice elements and their subset relationship as the partial order; alternatively, one could define a lattice for predicate abstraction using conjunctions over predicates from π as the lattice elements and their formula-implication relationship as partial order. The concretization function $[\![\cdot]\!]$ is defined by $[\![r]\!] = \{c \in C \mid c \models \varphi_r\}$.
2. The transfer relation $\rightsquigarrow_{\mathbb{P}}$ has the transfer $r \overset{g}{\rightsquigarrow}_{\mathbb{P}} r'$ if $\mathsf{post}(\varphi_r, g)$ is satisfiable and r' is the largest set of predicates from π such that φ_r implies $\mathsf{pre}(p, g)$ for each $p \in r'$, where $\mathsf{post}(\varphi, g)$ and $\mathsf{pre}(\varphi, g)$ denote the strongest post-condition and the weakest pre-condition, respectively, for a formula φ and a control-flow edge g. The two operators post and pre are defined such that $[\![\mathsf{post}(\varphi, g)]\!] = \{c' \in C \mid \exists c \in C : c \overset{g}{\rightarrow} c' \wedge c \models \varphi\}$ and $[\![\mathsf{pre}(\varphi, g)]\!] = \{c \in C \mid \exists c' \in C : c \overset{g}{\rightarrow} c' \wedge c' \models \varphi\}$. The Cartesian abstraction of the successor state is obtained by separate entailment checks for each predicate in π, which can be implemented by $|\pi|$ calls of a theorem prover.[11]

[11] A more efficient formulation of the same problem is based on the weakest pre-condition in order to avoid existential quantification.

3. The merge operator does not combine elements when the control flow meets: $\mathsf{merge}_{\mathbb{P}} = \mathsf{merge}^{sep}$.
4. The termination check considers abstract states individually: $\mathsf{stop}_{\mathbb{P}} = \mathsf{stop}^{sep}$.

Note that the CPA $\mathbb{P}$ cannot run alone: it is a component analysis that works in a composite analysis with the location analysis from Sect. 16.4.1 as another component CPA. The analysis could in principle be designed such that the predicates in π also constrain the program location, but this is not considered here.

The first practical implementations of a program analysis with Cartesian predicate abstraction were developed more than ten years ago (cf. SLAM [3, 4] and BLAST [13, 71]). More recent advancements in predicate analysis use Boolean abstraction [9] instead of Cartesian abstraction, and a complete temporal separation of the computation of the predicate abstraction for a formula from the computation of the strongest post-condition for a program operation [22]. An overview of Cartesian predicate abstraction is also given in Chap. 15, and of SAT solving in Chap. 9.

16.4.5 Explicit-Heap Analysis

In the following, we outline a simple analysis of dynamic data structures on the heap (as an extension of the simple programming language that we used so far), which is, for example, used as a basis for an accelerated abstraction in shape analysis [18, 19]; we give only a coarse overview here.

The *explicit-heap analysis* is a configurable program analysis $\mathbb{H} = (D_{\mathbb{H}}, \rightsquigarrow_{\mathbb{H}}, \mathsf{merge}_{\mathbb{H}}, \mathsf{stop}_{\mathbb{H}})$, which tracks explicit heap structures up to a certain size and consists of the following components:

1. The domain of the explicit-heap analysis stores concrete instances of data structures in its abstract states. Each abstract state represents an explicit, finite part of the memory. An *abstract state* $H = (v, h)$ of an explicit-heap analysis consists of the following two components: (1) the variable assignment $v : X \to \mathbb{Z}_{\top}$ is a total function that maps each variable identifier (integer or pointer variable) to an integer (representing an integer value or a structure address) or the special value $\top$ (representing the value "unknown"); and (2) the heap assignment $h : \mathbb{Z} \rightharpoonup (F \to \mathbb{Z}_{\top})$ is a partial function that maps every valid structure address to a field assignment, also called a *structure cell* (memory content). A field assignment is a total function that maps each field identifier $f \in F$ of the structure to an integer, or the special value $\top$. We call H an *explicit heap*. The initial explicit heap $H_0 = (v_0, \{\})$, with $v_0(x) = \top$ for every program variable x, represents all program states. Given an explicit heap H and a structure address a, the *depth* of H from a, denoted by $depth(H, a)$, is defined as the maximum length of an acyclic path whose nodes are addresses and where an edge from a_1 to a_2 exists if $h(a_1)(f) = a_2$ for some field f, starting from $v(a)$. The *depth* of H, denoted by $depth(H)$, is defined as $\max_{a \in X} depth(H, a)$.

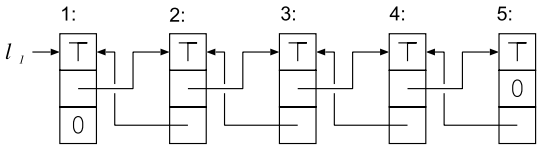


Fig. 3 Sample explicit heap for a doubly-linked list

2. The transfer relation $\rightsquigarrow_{\mathbb{H}}$ has the transfer $H \overset{g}{\rightsquigarrow}_{\mathbb{H}} H'$ if $H' = (v', h')$ is the explicit heap that results from applying the control-flow edge $g = (l, op, l')$ to the explicit heap $H = (v, h)$ according to the semantics of op. The new variable assignment v' maps every pointer variable p to $\top$ for which $depth(H, p) > c$, where c is an analysis-dependent constant maximal depth value of the heap structures.[12] (The analysis stops tracking structures that have a depth greater than the maximal depth value.)
3. The merge operator does not combine elements when the control flow meets: $\mathsf{merge}_{\mathbb{H}} = \mathsf{merge}^{sep}$.
4. The termination check considers abstract states individually: $\mathsf{stop}_{\mathbb{H}} = \mathsf{stop}^{sep}$.

Besides explicit-heap analysis, which can only serve as an auxiliary analysis or for bounded bug finding, several approaches for symbolic-heap analysis were proposed in the literature [16, 19, 32, 45, 75, 103].

Example 9 Figure 3 graphically depicts an explicit heap (v, h) that can occur in a program operating on a structure `elem{int data; elem* succ; elem* prev}`, with $v = \{l_1 \mapsto 1\}$ and $h = \{$
$1 \mapsto \{\texttt{data} \mapsto \top, \texttt{succ} \mapsto 2, \texttt{prev} \mapsto 0\},$
$2 \mapsto \{\texttt{data} \mapsto \top, \texttt{succ} \mapsto 3, \texttt{prev} \mapsto 1\},$
$3 \mapsto \{\texttt{data} \mapsto \top, \texttt{succ} \mapsto 4, \texttt{prev} \mapsto 2\},$
$4 \mapsto \{\texttt{data} \mapsto \top, \texttt{succ} \mapsto 5, \texttt{prev} \mapsto 3\},$
$5 \mapsto \{\texttt{data} \mapsto \top, \texttt{succ} \mapsto 0, \texttt{prev} \mapsto 4\}$
$\}$.

16.4.6 BDD Analysis

Binary decision diagrams (BDDs) [31] are a popular data structure in model-checking algorithms. In the following, we define a configurable program analysis that uses BDDs to represent abstract states. For the details, we refer to an article on the topic [25]. An introduction to BDDs is given in Chap. 7 and to BDD-based model checking in Chap. 8. Given a first-order formula φ over the set X of program variables, we use $\mathscr{B}_\varphi$ to denote the BDD that is constructed from φ, and $[\![\varphi]\!]$ to denote all variable assignments that fulfill φ. Given a BDD $\mathscr{B}$ over X, we use $[\![\mathscr{B}]\!]$ to denote all variable assignments that $\mathscr{B}$ represents ($[\![\mathscr{B}_\varphi]\!] = [\![\varphi]\!]$).

[12] The analysis has to apply garbage collection in heap assignments of the abstract states.

The *BDD analysis* is a configurable program analysis $\mathbb{BPA} = (D_{\mathbb{BPA}}, \rightsquigarrow_{\mathbb{BPA}}, \mathsf{merge}_{\mathbb{BPA}}, \mathsf{stop}_{\mathbb{BPA}})$ that represents the data states of the program symbolically, by storing the values of variables in BDDs. The CPA consists of the following components (taken from [25]):

1. The abstract domain $D_{\mathbb{BPA}} = (C, \mathscr{E}_{\widehat{\mathscr{B}}}, [\![\cdot]\!])$ is based on the semi-lattice $\mathscr{E}_{\widehat{\mathscr{B}}}$ of BDDs, i.e., every abstract state consists of a BDD. The concretization function $[\![\cdot]\!]$ assigns to an abstract state $\mathscr{B}$ the set $[\![\mathscr{B}]\!]$ of all concrete states that are represented by the BDD. Formally, the semi-lattice $\mathscr{E}_{\widehat{\mathscr{B}}} = (\widehat{\mathscr{B}}, \sqsubseteq, \sqcup, \top)$—where $\widehat{\mathscr{B}}$ is the set of all BDDs and $\top = \mathscr{B}_{true}$ is the BDD that represents all concrete states (1-terminal node)—is induced by the partial order $\sqsubseteq$ that is defined as: $\mathscr{B} \sqsubseteq \mathscr{B}'$ if $[\![\mathscr{B}]\!] \subseteq [\![\mathscr{B}']\!]$. The join operator $\sqcup$ for two BDDs $\mathscr{B}$ and $\mathscr{B}'$ yields the least upper bound $\mathscr{B} \vee \mathscr{B}'$.
2. The transfer relation $\rightsquigarrow_{\mathbb{BPA}}$ has the transfer $\mathscr{B} \overset{g}{\rightsquigarrow} \mathscr{B}'$ with

$$\mathscr{B}' = \begin{cases} \mathscr{B} \wedge \mathscr{B}_p & \text{if } g = (l, \texttt{assume}(p), l') \text{ and } [\![\mathscr{B} \wedge \mathscr{B}_p]\!] \neq \{\} \\ (\exists w : \mathscr{B}) \wedge \mathscr{B}_{w=e} & \text{if } g = (l, \mathtt{w} := \mathtt{e}, l'). \end{cases}$$

3. The merge operator is defined by $\mathsf{merge}_{\mathbb{BPA}} = \mathsf{merge}^{join}$.
4. The termination check is defined by $\mathsf{stop}_{\mathbb{BPA}} = \mathsf{stop}^{sep}$.

A complete program analysis can be instantiated by composing the CPA $\mathbb{BPA}$ for BDD-based analysis with the CPA $\mathbb{L}$ for location analysis, in order to also track the program locations.

16.4.7 Observer Automata

Many software verifiers require the user to encode the safety property to be verified as a reachability problem inside the program source code. It has been shown that tools can provide more convenient and elegant specification languages for expressing safety properties separately from the program [5, 8]. This approach has the advantages that the property need not be present in the program source code, and that different properties can be checked independently (and possibly simultaneously). The software model checker BLAST [8] provides a transformation technique that takes as input the original program and the specification, and produces an instrumented program. The instrumented program is then given to the standard model checker, which simply checks for reachability of an error label.

This approach can be realized even more elegantly using a composite analysis that transforms the specification into an observer automaton that runs in parallel with the other analyses of the verifier in a composition. Such a strategy was implemented, for example, in the software verifiers BLAST [120], CPACHECKER [21], and ORION [44].

A *specification* is an abstract description of a set of valid program paths for a given program. We represent such a specification as an *observer automaton* that

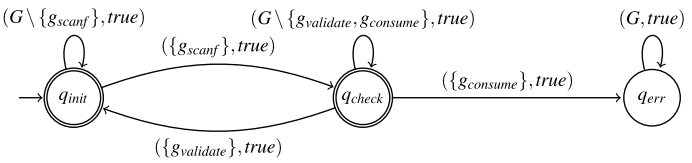


Fig. 4 Simple observer automaton

observes whether an invalid program path is encountered. Observer automata are also called "monitors" in the literature and can be generated from temporal-logic specifications. This idea is also used in test-case generation, where (temporal) coverage criteria are transformed into test-goal automata [20], and for re-playing error witnesses [11, 26].

Example 10 Consider as specification that each user input that the program reads (e.g., via `scanf`) must be validated by a call to a function `validate` before it is consumed (e.g., via function `consume`). The observer automaton in Fig. 4 starts in accepting state q_{init}, and switches to another accepting state q_{check} when `scanf` is called. From there, the automaton switches back to state q_{init} if function `validate` is called, but it switches to a non-accepting sink state if the input value is consumed without validation.

An *observer automaton* $A = (Q, \Sigma, \delta, q_{init}, F)$ for a CFA (L, l_0, G) is a non-deterministic finite automaton, with the finite set Q of control states, the alphabet $\Sigma \subseteq 2^G \times \Phi$ consisting of pairs that consist of a finite set of CFA edges and a state condition, the transition relation $\delta \subseteq Q \times \Sigma \times Q$, the initial control state $q_{init} \in Q$, and the set F of final control states (usually, all control states except the error control state $q_{err} \in Q$ are accepting control states, i.e., $F = Q \setminus \{q_{err}\}$). Let $p \in Q$ be the current state of an automaton A. The meaning of a transition $(p, (D, \psi), q) \in \delta$ is as follows: for a given control-flow edge g of the program analysis, the successor control state is control state q if the edge g matches one of the edges in the set D of edges. In combination with another CPA, using a strengthening operator, the successor state can be required to fulfill condition ψ (a later section will describe this). The observer automaton A accepts all program paths that have not reached the error control state, and rejects all program paths that reach the error control state. The specification that the observer automaton represents is fulfilled if all program paths are accepted by the observer automaton.

The *observer analysis* for an observer automaton A is a configurable program analysis $\mathbb{O} = (D_{\mathbb{O}}, \rightsquigarrow_{\mathbb{O}}, \mathsf{merge}_{\mathbb{O}}, \mathsf{stop}_{\mathbb{O}})$, that tracks the control state of the observer automaton $A = (Q, \Sigma, \delta, q_{init}, F)$, with $\Sigma \subseteq 2^G \times \Phi$, and consists of the following components (for a given CFA (L, l_0, G)):

1. The abstract domain $D_{\mathbb{O}} = (C, \mathscr{Q}, [\![\cdot]\!])$ consists of the set C of concrete states, the semi-lattice $\mathscr{Q}$, and a concretization function $[\![\cdot]\!]$. The semi-lattice $\mathscr{Q} = (Z, \sqsubseteq, \sqcup, \top_{\mathscr{Q}})$, with $Z = (Q \cup \{\top\}) \times \Phi$, consists of the set Z of abstract

data states, which are pairs of a control state from Q (or special lattice element) and a condition from Φ, a partial order $\sqsubseteq$, the join operator $\sqcup$, and the top element $\top_{\mathcal{D}}$. The partial order $\sqsubseteq$ is defined such that $(q, \psi) \sqsubseteq (q', \psi')$ if $(q' = \top$ or $q = q')$ and $\psi \Rightarrow \psi'$, the join $\sqcup$ is the least upper bound of two abstract data states, and the top element $\top_{\mathcal{D}} = (\top, \mathit{true})$ is the least upper bound of the set of all abstract data states. The concretization function $[\![\cdot]\!] : Z \to 2^C$ is a mapping that assigns to each abstract data state (q, ψ) the set $[\![\psi]\!]$ of concrete states.

2. The transfer relation $\rightsquigarrow_{\mathbb{O}}$ has the transfer $(q, \psi) \overset{g}{\rightsquigarrow}_{\mathbb{O}} (q', \psi')$ if the observer automaton A has a transition $(q, (D, \psi'), q') \in \delta$ such that $g \in D$. The condition ψ' of the state transition is stored in the successor in order to enable a composite strengthening operator to strengthen the successor abstract data state of another component analysis in the composite analysis using information from condition ψ'.
3. The merge operator combines elements with the same control state:

$$\mathsf{merge}_{\mathbb{O}}\big((q, \psi), (q', \psi')\big) = \begin{cases} (q', \psi \vee \psi') & \text{if } q = q' \\ (q', \psi') & \text{otherwise.} \end{cases}$$

4. The termination check considers control states and conditions of the automaton individually: $\mathsf{stop}_{\mathbb{O}} = \mathsf{stop}^{sep}$.

16.5 Combination Examples (Composite Analyses)

We describe several examples in which component analyses are assembled into composite analyses that are neither pure data-flow analyses nor pure model checking. We show that such combinations are relatively easy to express in the CPA formalism, and explain what is taken from which approach and why it is useful to combine them.

16.5.1 Predicate Analysis + Constant Propagation

"Predicated lattices" are a practical combination of the predicate-abstraction domain with a classic data-flow domain [49]. The predicate analysis behaves as in model checking: abstract predicate states are never joined. The composite analysis performs a merge of two composite abstract states as follows: if the two component abstract states of the predicate analysis are equal, then the two component abstract states of the data-flow analysis are joined and the composite analysis stores one composite abstract state, otherwise the composite analysis stores two separate composite abstract states.

Given the CPA $\mathbb{P}$ for predicate analysis and any CPA for data-flow analysis, for example, the CPA $\mathbb{CO}$ for constant propagation. The composite program analysis $\mathscr{C}_{\mathbb{PCO}} = (\mathbb{L}, \mathbb{P}, \mathbb{CO}, \rightsquigarrow_\times, \mathsf{merge}_\times, \mathsf{stop}_\times)$ for a predicated constant propagation consists of the following components: the CPA $\mathbb{L}$ for location tracking from Sect. 16.4.1, the CPA $\mathbb{P}$ for predicate analysis from Sect. 16.4.4, the CPA $\mathbb{CO}$ for constant propagation from Sect. 16.4.2, the composite transfer relation $\rightsquigarrow_\times$, the composite merge operator $\mathsf{merge}_\times$, and the composite termination check $\mathsf{stop}_\times$. The composite transfer relation $\rightsquigarrow_\times$ has the transfer $(e_1, e_2, e_3) \overset{g}{\rightsquigarrow}_\times (e'_1, e'_2, e'_3)$ if $e_1 \overset{g}{\rightsquigarrow}_{\mathbb{L}} e'_1$ and $e_2 \overset{g}{\rightsquigarrow}_{\mathbb{P}} e'_2$ and $e_3 \overset{g}{\rightsquigarrow}_{\mathbb{CO}} e'_3$. The composite merge operator $\mathsf{merge}_\times$ is defined by

$$\mathsf{merge}_\times\big((e_1, e_2, e_3), \big(e'_1, e'_2, e'_3\big)\big) = \begin{cases} (e_1, e_2, \mathsf{merge}_{\mathbb{CO}}(e_3, e'_3)) & \text{if } e_1 = e'_1 \text{ and } e_2 = e'_2 \\ (e'_1, e'_2, e'_3) & \text{otherwise.} \end{cases}$$

The composite termination check is defined by $\mathsf{stop}_\times = \mathsf{stop}^{sep}$.

For the combination of predicate analysis with a data-flow analysis for pointers, it has been shown that this configuration can significantly improve the verification performance [49].

16.5.2 Predicate Analysis + Constant Propagation + Strengthen

Now we extend the above composite program analysis by using a strengthening operator in the transfer relation. Again, a strengthening operator $\downarrow : E_1 \times E_2 \to E_1$ takes an abstract state $e_1 \in E_1$ as input and uses information stored in an abstract state $e_2 \in E_2$ from another CPA to constrain ("strengthen") the set of concrete states that the resulting abstract state $\downarrow(e_1, e_2)$ represents.

We use a strengthening operator of the concrete type $\downarrow_{\mathbb{CO},\mathbb{P}} : E_{\mathbb{CO}} \times E_{\mathbb{P}} \to E_{\mathbb{CO}}$, i.e., it strengthens a variable assignment from the constant propagation with an abstract state from the predicate analysis (set of predicates that are satisfied). The strengthening operator $\downarrow_{\mathbb{CO},\mathbb{P}}(v, r)$ is defined, if $\phi_v \wedge \phi_r$ is satisfiable, as follows, for every variable x:

$$\downarrow_{\mathbb{CO},\mathbb{P}}(v, r)(x) = \begin{cases} c & \text{if } c \text{ is the only satisfying assignment of } \phi_v \wedge \phi_r \text{ for } x \\ v(x) & \text{otherwise} \end{cases}$$

where $\phi_v := \bigwedge_{x \in X, v(x) \neq \top_{\mathscr{Z}}} x = v(x)$.

We now define the transfer relation for the new composite program analysis: The composite transfer relation $\rightsquigarrow_\times$ has the transfer $(e_1, e_2, e_3) \overset{g}{\rightsquigarrow}_\times (e'_1, e'_2, e'_3)$ if $e_1 \overset{g}{\rightsquigarrow}_{\mathbb{L}} e'_1$, and $e_2 \overset{g}{\rightsquigarrow}_{\mathbb{P}} e'_2$, and $e_3 \overset{g}{\rightsquigarrow}_{\mathbb{CO}} e''_3$, and $\downarrow(e''_3, e'_2)$ is defined, and $e'_3 = \downarrow(e''_3, e'_2)$.

This combined analysis is more precise than the component analyses alone, which will be illustrated in the following example. A more flexible extension of this combination was presented using the concept of *dynamic precision adjustment* [18]. Experiments with this extension have shown that combinations with strengthening

```
1  int foo(int y) {
2    int x = 1;
3    int z = 1;
4    if (y < 1) {
5      return 0;
6    }
7    if (y > 1) {
8      x = 5;
9    } else {
10     z = 5 * x * y;
11   }
12   return 10 / (x - z);
13 }
```

Fig. 5 Example C function, used to illustrate predicated lattices with strengthening. Neither predicate analysis, nor constant propagation, nor a predicated lattice without strengthening are precise enough to prove the correctness of this function, but the combination with strengthening is

operators can be more effective and more efficient than Cartesian products of analyses. While the effects of such "reduced products" [39] have been known for decades, the framework of configurable program analysis enables us to express such combinations in a simple and elegant implementation.

Example 11 Consider the example program in Fig. 5, which extends the previous example with a non-linear expression. The safety property to be checked is that no division by zero is executed. Suppose we use a predicate analysis for the theory of linear arithmetics (LA) and equalities with uninterpreted functions (EUF), with the precision (i.e., set of predicates to track) $\{x = 1, z = 1, z = 5, y \geq 1, y \leq 1\}$ and a constant-propagation analysis.

The predicate analysis with location tracking does not succeed in proving this example safe: At program location 7, we have the predicate abstract data state $x = 1 \ \wedge \ z = 1 \ \wedge \ y \geq 1$. At program location 10, we have the abstract data state $x = 1 \ \wedge \ z = 1 \ \wedge \ y \geq 1 \ \wedge \ y \leq 1$. At program location 11, however, we have no information about z, due to the fact that the non-linear operation "*" is modeled as an uninterpreted function, resulting in the following abstract data state: $x = 1 \wedge y \geq 1 \wedge y \leq 1$. Thus, the analysis conservatively assumes that the program can fail with a division by zero. This cannot be remedied by adding other predicates.

The constant propagation stores the value $\top$ for program variable y after the assume operations from the `if` statements in lines 4 and 7, and thus, also cannot determine the value of z before the division is computed, and conservatively reports that the division might fail.

Also the "predicated lattice" (without strengthening) from Sect. 16.5.1 is not precise enough to prove that a division by zero cannot occur. Although the analysis is now path-sensitive with respect to the predicates, the constant propagation (which can precisely interpret the multiplication) cannot determine the value of program variable y, and the predicate analysis cannot determine the result of the multiplication, regardless of the predicates used.

The composite analysis with strengthening can transfer information from the predicate abstract data states to the variable assignments of the constant-

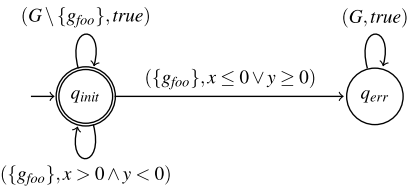


Fig. 6 Example observer automaton with conditions

propagation analysis. At program location 10, we have the abstract data state $\mathtt{x} = 1 \ \wedge \ \mathtt{z} = 1 \ \wedge \ \mathtt{y} \geq 1 \ \wedge \ \mathtt{y} \leq 1$ for the predicate analysis and $\{\mathtt{x} \mapsto 1, \mathtt{y} \mapsto \top, \mathtt{z} \mapsto 1\}$ for the constant-propagation analysis. Now, the composite analysis does not store the direct product of these two abstract data states as a composite abstract state, but first strengthens the variable assignment, in particular, of variable `y`: the only value for `y` that satisfies the predicate abstract data state is 1, and therefore, the new variable assignment after strengthening is $\{\mathtt{x} \mapsto 1, \mathtt{y} \mapsto 1, \mathtt{z} \mapsto 1\}$. The constant-propagation analysis can now compute a value not equal to $\top$ for program variable `z` for the assignment from location 10 to 11, and thus, is able to prove the program correct, i.e., that there is no division by zero.

16.5.3 Predicate Analysis + Explicit-Heap Analysis

Similarly to the analysis that incorporates the results of an inexpensive constant-propagation analysis into a predicate analysis, we can enhance the analysis by using the result of the explicit-heap analysis. Of course, in order to keep the analysis practically relevant, the threshold for the heap analysis should be small. This way, information about the heap that is normally not tracked by the predicate analysis can be fed to the predicate analysis via a composite strengthening operator, in order to make the path decisions of the predicate analysis more precise. A combination of an abstract domain with an explicit-heap analysis was used already in a different context [19], where a symbolic abstract shape representation was extracted from the explicit-heap results. The combination was shown to be able to verify more programs than the component analyses alone. This direction of combinations of program analyses is largely unexplored in the literature.

16.5.4 Predicate Analysis + Observer Automata

The following example observer automaton contains conditions at the transitions, but the observer analysis from Sect. 16.4.7 is not able to respect the conditions. After the motivating example, we introduce a combination analysis that is able to consider the conditions during the state-space exploration.

Example 12 Let us consider a specification that requires the pre-condition $x > 0 \wedge y < 0$ to be fulfilled whenever function `foo` is called. Figure 6 shows an observer

automaton for this specification, with initial state q_{init} and error state q_{err}. The observer automaton starts in control state q_{init}. As long as the exploration of the program encounters only control-flow edges different from g_{foo}, the automaton stays in control state q_{init}. Once a control-flow edge g_{foo} is taken in the exploration, our observer automaton has to consider the conditions at the transitions for g_{foo}: if the condition $\psi = x > 0 \wedge y < 0$ is fulfilled (specification satisfied) then the automaton stays in (accepting) control state q_{init}, otherwise the observer automaton switches to (non-accepting) control state q_{err}. The error state q_{err} is a sink state and thus the explored program path will be rejected (because it violates the specification).

We construct a composite program analysis that runs both the predicate analysis and the observer analysis as components. The resulting program analysis combines the abstract data states in such a way that (1) it uses information from the predicate analysis to determine the actual transition switch of the observer automaton, and (2) it marks all paths through the program that violate the specification with the control state q_{err}:

1. The composite domain $D_\times = D_\mathbb{P} \times D_\mathbb{O}$ is the product of the component domains $D_\mathbb{P}$ for the predicate analysis and $D_\mathbb{O}$ for the observer analysis.
2. The transfer relation $\rightsquigarrow_\times$ has the transfer $(\varphi, (q, \psi)) \overset{g}{\rightsquigarrow}_\times (\varphi', (q', \psi'))$ if $\varphi \overset{g}{\rightsquigarrow}_\mathbb{P} \varphi'$, and $(q, \psi) \overset{g}{\rightsquigarrow}_\mathbb{O} (q', \psi')$, and $\downarrow_{\mathbb{P},\mathbb{O}} (\varphi', (q', \psi'))$ is defined. The strengthening operator $\downarrow_{\mathbb{P},\mathbb{O}}$ is defined only if $\varphi' \wedge \psi'$ is satisfiable, in which case it returns $\varphi' \wedge \psi'$ as the abstract data state of the predicate analysis. In other words, the strengthening operator (a) eliminates successors of the observer automaton with conditions that contradict the abstract state of the predicate analysis and (b) restricts the abstract state of the predicate analysis to those concrete states that satisfy the condition of the observer automaton. The strengthening operator is necessary because both (a) and (b) can be evaluated only after the successors of all participating CPAs are known.
3. The merge operator keeps different abstract states separate: $\mathsf{merge}_\times = \mathsf{merge}^{sep}$.
4. The termination check considers abstract states individually: $\mathsf{stop}_\times = \mathsf{stop}^{sep}$.

Note on Soundness. The strengthening operator might replace the original abstract state by a new abstract state that represents fewer concrete states. To guarantee soundness of the composition program analysis, the observer automaton must not restrict the program exploration of other analyses, i.e., for all control states $q \in Q$, the disjunction of the conditions ψ_i' of all transitions $(q, (D_i, \psi_i'), q_i)$ that leave control state q must be equivalent to *true*.[13]

There are applications for which the soundness requirement is not desirable and the automata are used to control (restrict) the program exploration of the other analyses, for example, as used in test-goal automata [20] and error-witness automata [11, 26].

[13]This soundness requirement is easy to fulfill on the syntactical level by using a SPLIT operation in the definition of transitions of the automaton. The transition syntax SPLIT($x > 0 \wedge y < 0$, q_{init}, q_{err}), for example, defines the two transitions from control state q_{init} to q_{init} and to q_{err} (cf. Fig. 6).

16.6 Algorithms for Constructing Program Invariants

While the previous section focuses on abstract domains and how to practically combine abstract domains from data-flow analysis with abstract domains from model checking in a unifying, configurable framework, this section discusses different algorithmic styles that are used to compute program invariants in data-flow analysis and model checking.

The general idea is to construct a witness that proves the correctness or incorrectness of the program. To show that a program violates a property, an *error path* (*counterexample* [34], exchangeable *error witness* [11]) is constructed. To show that a program satisfies a property, a *program invariant* (main ingredient of a *correctness proof*) is constructed; program invariants can be stored as *certificates* [70] or exchangeable *correctness witnesses* [10]. The program invariants look different depending on the algorithm that is used to construct it.

Both data-flow analysis and model checking construct over-approximations of the reachable concrete states (Algorithm 1). For data-flow analysis, the program invariant is a function reached, which assigns to each reachable program location an over-approximation of all concrete data states that can occur at that location. For model checking, the program invariant is a set reached, which contains a set of abstract states that contain all concrete states of the program (possibly many abstract states for the same program location, depending on how path-sensitive the analysis is).

If the program invariant is computed via a fixed-point iteration, then the program invariant is called the *solution for the fixed-point problem*. In the following, we describe different algorithmic approaches to compute program invariants (fixed points).

16.6.1 Iterative and Monotonic Fixed-Point Approaches

The most commonly known and used algorithm for computing a program invariant consists of an iteration that initializes the unknown program invariants to a lower bound (or upper bound) of the abstract values and then updates them monotonically to compute a least (or greatest) fixed point over the underlying abstract domain or invariant language (cf. Sects. 16.2.2 and 16.3.2). This technique is used in various standard approaches to data-flow analysis and model checking. One notable dimension for analyses in this category is whether the analysis is forward or backward.

Forward analyses start from pre-conditions and propagate them forward (iteratively across loops until a fixed point is reached) to compute invariants at various program locations. Forward analyses have the advantage of not requiring the code to be annotated with post-conditions and hence can generate invariants not only for program verification but also for applications such as compiler optimization. The key challenge in designing a forward analysis is to design abstract transfer relations and merge operators (including *join* and *widen*) that can compute

over-approximations of strongest post-conditions (cf. Sect. 16.3.5). Such transfer relations are known for a variety of abstract domains, including linear arithmetic [41, 92], uninterpreted functions [57], combination of linear arithmetic and uninterpreted functions [61], heap-shape domains [16, 45, 103], combination of arithmetic and heap-shape domains [55], and quantified array properties [56].

Backward analyses typically require post-conditions to start with, but have the advantage of being goal-directed. The key challenge in backward analysis is to have an *abduct* procedure for computing under-approximations of weakest preconditions (as opposed to forward abstract transfer relations, which perform over-approximations of strongest post-conditions). Such procedures are known for some abstract domains including linear arithmetic [62], uninterpreted functions [62], combinations of linear arithmetic and uninterpreted functions [60], and heap shapes [32]. Backward analyses have received less attention in terms of research projects compared to forward analyses, but become more important in the context of combination of forward and backward reachability analysis [119].

Another notable dimension for analyses in this category is the order of iteration (cf. Sect. 16.3.5). In Algorithm 2, the iteration order is encoded in the operator choose, which selects the next abstract state to explore from the set waitlist of abstract states that are still to be processed. Besides the two simple graph-traversal orders depth-first search (DFS) and breadth-first search (BFS), there are many possible implementations of the choose operator, such as random order, chaotic order [29], post-order, reverse post-order [22], topological order, and many more (e.g., [88]).

16.6.2 Counterexample-Guided Abstraction Refinement

One of the challenges in data-flow analysis and model checking is to automatically construct an abstract model of a program, or more precisely, the level of abstraction for a given abstract domain. Many classic analyses hard-wire the abstraction level into the abstract domain, but the *precision* of the analysis can also be treated as a separate concern [18]. For example, if the abstract domain is taken from constant propagation, then the precision can be a set of variables and determine which variables are tracked; if the abstract domain is predicate abstraction, the precision is the set of predicates that are tracked.

The problem of computing an appropriate precision can be solved by counterexample-guided abstraction refinement (CEGAR) [34]. This technique works orthogonally to the above-mentioned iteration techniques. The analysis approach (e.g., iterative) starts with a coarse precision (very abstract model), and successively refines the abstract model by adding information to the precision. If the analysis finds a violation of the property to be verified, then the abstract error path is analyzed. If the abstract error path represents a concrete error path (executable violation), then the analysis can stop and report the violation. If the abstract error path does not represent a concrete error path (infeasible path) then that path was found due to a too-coarse (too imprecise) abstract model, and the abstract error path can be used

to find out what information is necessary to track in the abstract model in order to eliminate this abstract error path from further explorations. The extracted information is added to the precision, the set of reached abstract states is updated, and the analysis continues.

While CEGAR is most popular for predicate analysis, the technique has also been explored for value analysis [24, 96] and symbolic execution [23]. There are several techniques to extract information from counterexamples, for example, extraction from syntax and weakest pre-conditions using a set of heuristics [4, 13, 21, 35], using Craig interpolation [13, 21, 42, 69], and using invariant synthesis [15]. More details are provided in Chap. 14 on interpolation and in Chap. 13 on CEGAR.

16.6.3 Template- and Constraint-Based Approaches

Constraint-based approaches construct the program invariant by guessing a second-order template for each necessary loop invariant such that the only unknowns in the second-order template are first-order quantities. Then, the approach generates constraints over those first-order unknowns (after substituting the guessed form into the program invariant). The generated constraints are existentially quantified in the first-order unknowns, but universally quantified in the program variables. The challenge of solving these constraints is to have a procedure to eliminate the universally quantified variables from the constraints, and then solve the constraints for the existentially quantified variables by using some off-the-shelf constraint solver.

Consider the following example program:

```
1 if (n <= 0) { return 1;}
2 x = 0; y = 1;
3 while (x != n) { x = x + 1; y = y + 2; }
4 assert(y == 2n + 1);
```

Suppose we guess that the loop invariant that is required to prove the assertion is of the form $a\mathtt{x} + b\mathtt{y} + c\mathtt{n} + d = 0$, where a, b, c, and d are unknown integer constants. Substituting this loop-invariant template into the program invariant for the above program yields the following constraints for the loop head, where all program variables `x`, `y`, and `n` are universally quantified:

$$\mathtt{n} > 0 \ \wedge \ \mathtt{x} = 0 \ \wedge \ \mathtt{y} = 1 \ \Rightarrow \ a\mathtt{x} + b\mathtt{y} + c\mathtt{n} + d = 0$$

$$a\mathtt{x} + b\mathtt{y} + c\mathtt{n} + d = 0 \ \wedge \ \mathtt{x} = \mathtt{n} \ \Rightarrow \ \mathtt{y} = 2\mathtt{n} + 1$$

$$a\mathtt{x} + b\mathtt{y} + c\mathtt{n} + d = 0 \ \wedge \ \mathtt{x} \neq \mathtt{n} \ \Rightarrow \ (a\mathtt{x} + b\mathtt{y} + c\mathtt{n} + d = 0)_{[\mathtt{x} \mapsto (\mathtt{x}+1), \mathtt{y} \mapsto (\mathtt{y}+2)]}.$$

Farkas' lemma can be used to eliminate universally quantified variables from the above constraint, thereby obtaining the following constraint:

$$c = 0 \ \wedge \ b + d = 0 \ \wedge \ 2b + a + c = 0 \ \wedge \ b \neq 0.$$

An off-the-shelf first-order constraint solver may now generate the solution $a = -2, b = 1, d = -1$, thereby yielding the invariant $y = 2x + 1$.

This kind of invariant-computation technique has been developed for a variety of abstract domains including linear inequalities [104], disjunctions of linear inequalities [58], non-linear inequalities [63, 105], combination of linear inequality and uninterpreted functions [14], predicate abstraction [15, 59], and quantified invariants [111]. The key component in the algorithms for these domains is often a novel procedure to eliminate universal quantification.

16.6.4 Proof-Rule-Based Approaches

Approaches that are based on proof rules require the analysis designer to have a good understanding of the design patterns (for loop behaviors) that occur in practice, and then to develop proof rules for each of these design patterns. The beauty of this approach is that it usually enables the analysis of program loops by simply reasoning about their (loop-free) bodies in order to identify the appropriate design pattern and apply the corresponding rule. The reasoning about loop-free code fragments can be done using off-the-shelf SMT solvers. This approach has been applied for a variety of program analyses, including symbolic computational-complexity analysis [64], continuity analysis [33], and variable-bound analysis [54].

Example 13 If the transition system of a loop implies that $\mathtt{x}' = \mathtt{x} << 1 \ \wedge \ \mathtt{x} \neq 0$ or $\mathtt{x}' = \mathtt{x} \& (\mathtt{x} - 1) \ \wedge \ \mathtt{x} \neq 0$, then $LSB(\mathtt{x})$ is a ranking function for that loop (where $\mathtt{x}$ is any loop variable, $\mathtt{x}'$ denotes the update to that loop variable, and $LSB(\mathtt{x})$ returns the least significant bit of $\mathtt{x}$, and $<<$ and & represent bitwise-left-shift and bitwise-and operators respectively) [64]. As another example, if the transition system of a loop is of the form $s_1 \vee s_2$, and r_1 and r_2 are ranking functions for s_1 and s_2, respectively, and s_1 (resp., s_2) implies that r_2 (resp., r_1) is non-increasing, then the number of iterations of the loop above is bounded by $\mathtt{Max}(0, r_1) + \mathtt{Max}(0, r_2)$ [64]. Note that these judgments about loop properties require discharging standard SMT queries that are constructed using transition systems that represent loop-free code fragments.

16.6.5 Iterative, but Non-monotonic Approaches

There are techniques for computing fixed points that are iterative and converging, but have non-monotonic progress towards a fixed-point solution [50]. In each of the two techniques that we explain below, the non-monotonic iteration has the unifying property that the distance between the iterated abstract states and a fixed-point solution (according to some underlying distance measure) decreases in each iteration (as in the case of Newton's method for computing the roots of an equation).

Probabilistic Inference. Inspired by techniques from machine learning, we can pose the problem of computing a program invariant as an inference in probabilistic graph

models, which allows the use of probabilistic inference techniques like belief propagation, in order to perform the fixed-point computation. This technique requires us to develop appropriate distance measures between any two abstract states of an abstract domain (which traditionally is equipped with only a partial order) to guide the progress of the probabilistic inference algorithm. This technique has been applied to discovering disjunctive quantifier-free invariants on numerical programs [53]. The algorithm iteratively selects a program location randomly and updates the current abstract state to make it more *locally consistent* with respect to the abstractions at the neighboring program locations (as per the underlying distance measure, until convergence). Interestingly, this simple algorithm was shown to converge in a few rounds for the chosen benchmark examples, yielding the desired invariants. The distance measure, for a pair of abstract states e_1 and e_2, was chosen to be proportional to the number of pairs (i, j) such that e_1^i does not imply e_2^j, where $\bigvee_{i=1}^{n} e_1^i$ is the disjunctive normal form representation of e_1 and $\bigwedge_{j=1}^{m} e_2^j$ is the conjunctive normal form representation of e_2. (Note that if e_1 implies e_2 then each e_1^i implies each e_2^j.) Observe that the local inconsistency of the abstract state at a program location is thus a monotonic measure of the set of abstract states that are not consistent with the abstract states at the neighboring program locations.

Learning. Inspired by techniques from concept learning, we can pose the problem of computing a program invariant as an instance of algorithmic learning that requires an oracle to answer simpler questions about the invariant, such as whether a given invariant is the desired one (equivalence question), or whether a given state is a model of the desired invariant (membership question). This technique has been applied to discover quantifier-free invariants [76] as well as quantified invariants [83].

16.6.6 Comparison with Standard Recurrence Solving

The three techniques "iterative monotonic," "template-based," and "proof-rule-based" for computing program invariants bear striking similarity to the three standard techniques that have long been known in the area of algorithms for generating closed form upper/lower approximations for recurrences [37]. A *recurrence* is an equation or inequality that describes a function in terms of its value on smaller inputs. Recurrences are useful to describe the run time of recursive algorithms.

Substitution Method. The substitution method guesses the template of the solution and then generates constraints (over the first-order unknowns in the guessed template) after substituting the guessed template into the recurrence relation. Any solution to the generated constraints is a valid solution to the recurrence relation. This method is powerful, but requires a template of the answer to be guessed, which takes experience and sometimes requires creativity.

Example 14 Consider the problem of computing a closed form solution for the recurrence $T(n) = T(n-1) + 6n$ with the boundary condition $T(1) = 2$. Suppose we guess that the solution is of the form $T(n) = an^3 + bn^2 + cn + d$, for some (unknown) constants a, b, c, and d. Substituting the guessed form into the recurrence relation and the boundary conditions yields the following constraints:

$$\begin{aligned} an^3 + bn^2 + cn + d &\equiv a(n-1)^3 + b(n-1)^2 + c(n-1) + d + 6n, \\ a + b + c + d &\equiv 2. \end{aligned}$$

These constraints imply $a = 0$, $c = b = 3$, and $d = -4$. This yields the solution $T(n) = 3n^2 + 3n - 4$.

Iteration Method. The idea of the iteration method is to expand (iterate) the recurrence and express it as a sum of terms that are dependent only on n and the initial conditions. Techniques for evaluating sums can then be used to provide bounds on the solution.

Example 15 Consider the recurrence $T(n) = 3T(\lfloor n/4 \rfloor) + n$. We iterate it and then express it as a summation as follows (using the observation that the sub-problem size hits $n = 1$ when i exceeds $\log_4 n$):

$$\begin{aligned} T(n) &= n + 3\,T\big(\lfloor n/4 \rfloor\big) \\ &= n + 3\big(\lfloor n/4 \rfloor + 3\,T\big(\lfloor n/16 \rfloor\big)\big) \\ &= n + 3\big(\lfloor n/4 \rfloor + 3\big(\lfloor n/16 \rfloor + 3\,T\big(\lfloor n/64 \rfloor\big)\big)\big) \\ &= n + 3\,\lfloor n/4 \rfloor + 9\,\lfloor n/16 \rfloor + 27\,T\lfloor n/64 \rfloor \\ &\leq n + 3n/4 + 9n/16 + 27n/64 + \cdots + 3^{\log_4 n}\Theta(1) \\ &\leq n \sum_{i=0}^{\infty} (3/4)^i + \Theta\big(n^{\log_4 3}\big) \\ &= 4n + o(n) \\ &= \mathrm{O}(n). \end{aligned}$$

Master Method. The master method relies on the following theorem, which provides a case-based method for solving recurrences of the form $T(n) = aT(n/b) + f(n)$.

Theorem 1 (Master Theorem [37]) *Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the non-negative integers by the recurrence*

$$T(n) = aT(n/b) + f(n)$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then, $T(n)$ can be bounded asymptotically as follows:

1. *If* $f(n) = \mathrm{O}(n^{\log_b a - \epsilon})$ *for some constant* $\epsilon > 0$, *then* $T(n) = \Theta(n^{\log_b a})$.
2. *If* $f(n) = \Theta(n^{\log_b a})$ *then* $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. *If* $f(n) = \Omega(n^{\log_b a + \epsilon})$ *for some constant* $\epsilon > 0$, *and if* $a \cdot f(n/b) \leq c \cdot f(n)$ *for some constant* $c < 1$ *and all sufficiently large* n, *then* $T(n) = \Theta(f(n))$.

Example 16 (Using the Master Method [37]) To use the master method, we simply need to determine which case (if any) of the master theorem matches the given recurrence relation. For example, for the recurrence $T(n) = 9T(n/3) + n$, we have $a = 9, b = 3$, $f(n) = n$, and thus $n^{\log_b a} = \Theta(n^2)$. Since $f(n) = \mathrm{O}(n^{\log_3 9 - \epsilon})$, where $\epsilon = 1$, we can apply Case 1 of the master theorem to conclude that $T(n) = \Theta(n^2)$.

Connections. The substitution method for solving recurrences is quite similar to the template-based method for program invariant generation. The iteration method for solving recurrences is similar to the iterative monotonic techniques for invariant generation in that both require iteration (or unrolling) of the underlying recursive system of equations to perform an appropriate generalization. The master method for solving recurrences is in the same category as the proof-rule-based method for invariant generation since both involve (manually) establishing non-trivial theorems to allow easy automated reasoning of most instances by simply requiring a matching engine to match the given instance against an existing small collection of general rules. It is heartening to observe that two different communities have ended up discovering similar classes of useful techniques for reasoning about recursive systems!

16.6.7 Discussion

We now briefly discuss the advantages and disadvantages of the different techniques.

The iterative monotonic techniques have been the most popular choice in the domains of data-flow analysis and model checking, primarily because they are the oldest and most well-understood techniques. These techniques have also been very successful because they generally allow for selecting the right trade-off between precision and scalability.

The CEGAR algorithm is popular mainly in model checking; it is not applicable to path-insensitive data-flow analysis, because if a property violation is found, then an error path needs to be constructed. The technique is orthogonal to the algorithm that constructs the program invariant—it only requires a notion of precision in order to determine and adjust the abstraction level of the analysis.

Template-based techniques are generally the least scalable because they often involve the use of sophisticated constraint solvers; they are not successful in practice if used in isolation because of the scaling problem. However, these techniques are most effective in analyzing sophisticated properties of small programs, and can be practicable if applied to smaller sub-problems in a larger verification setting, such as computing invariants for path programs [15] during the verification of large programs. The techniques also have further enabled synthesis of small programs [74, 112].

Proof-rule-based techniques are the most scalable, and have been applied to the analysis of large programs; they are limited in applicability because they require the existence and knowledge of a small set of design patterns that occur in the programs to be analyzed. When applicable, proof-rule-based techniques might be the best choice (just as the master method is the most popular choice for analyzing the run time of standard recursive algorithms as found in textbooks).

Probabilistic inference and learning-based techniques are not yet widely used, and it remains to be seen whether they can produce new impactful results in the area of program verification. Their true strength may lie in dealing with noisy or under-specified systems, and especially in the synthesis of systems.

16.7 Combinations in Tool Implementations

During recent years, combining approaches from data-flow analysis and model checking became state-of-the-art in tool implementations. To witness this development, we give an overview of the techniques and features that modern software verifiers implement. As a reference collection of tools for software verification, we refer to the Competition on Software Verification. In 2014, a total of 15 verifiers participated in the competition (including demo track); detailed results are available in the competition report [6] and on the competition web site.[14]

Table 1 lists the features and technologies that are used in the verification tools. This illustrates that techniques from data-flow analysis and model checking are combined to achieve better results: Counterexample-guided abstraction refinement (CEGAR, cf. Chap. 13, [34]), predicate abstraction (cf. Chap. 15, [52]), bounded model checking (BMC, cf. Chap. 10, [27]), abstract reachability graphs (ARGs, cf. [13]), lazy abstraction (cf. [16, 71]), interpolation for predicate refinement (cf. Chap. 14, [69]), and termination checking via ranking functions (cf. Chap. 15, [98]) are typical examples of techniques contributed by the model-checking community. Value analysis (similar to constant propagation, cf. [24]), interval analysis (cf. [94]), and shape analysis (cf. [32, 45, 75, 103]) are typical examples of abstract domains from the data-flow community.

16.8 Conclusion

In theory, there is no difference in expressive power between data-flow analysis and model checking. This chapter describes the paradigmatic and practical differences of the two approaches, which are relevant especially for precision and performance characteristics. The unifying formal framework of configurable program

[14] http://sv-comp.sosy-lab.org/.

Table 1 Techniques that current verification tools implement (adapted from [6])

Verification tool	CEGAR	Predicate abstraction	Bounded model checking	Explicit-value analysis	Interval analysis	Shape analysis	ARG-based analysis	Lazy abstraction	Interpolation	Ranking functions
APROVE [51]										✓
BLAST [13, 109]	✓	✓					✓	✓	✓	
CBMC [85]			✓							
CPALIEN [91]				✓		✓				
CPACHECKER [21, 87]	✓	✓	✓	✓	✓	✓	✓	✓	✓	
CSEQ [72, 117]			✓							
ESBMC [90]			✓							
FUNCTION [118]					✓					✓
FRANKENBIT [66]			✓						✓	
LLBMC [48]			✓							
PREDATOR [46]						✓				
SYMBIOTIC [110]										
T2 [30]	✓	✓			✓		✓	✓	✓	✓
TAN [84]	✓	✓	✓		✓			✓		✓
THREADER [99]	✓	✓					✓		✓	
UFO [65]	✓	✓	✓		✓		✓	✓	✓	
ULTIMATE [47, 67, 68]	✓	✓						✓	✓	✓

analysis makes the differences explicit. This framework enables an easy combination of abstract domains, no matter whether they were invented for data-flow analysis or for model checking. Several examples demonstrate that the combination of abstract domains designed for data-flow analysis with abstract domains designed for software model checking improves both effectiveness (precision) and efficiency (performance) of such analyses. The new, configurable combinations make it possible to plug together composite program analyses that are strictly more powerful than the component analyses. The chapter also provides an overview of the different flavors of algorithms for computing the same solution: program invariants. There are several different approaches, originating from different research communities, and combinations have a large potential for further improving the state of the art. Modern tools for software verification—as witnesses of our considerations—almost always combine techniques from data-flow analysis with techniques from model checking.

References

1. Abadi, M., Cardelli, L.: A Theory of Objects. Springer, Heidelberg (1996)
2. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley, Reading (1986)
3. Ball, T., Podelski, A., Rajamani, S.K.: Boolean and Cartesian abstractions for model checking C programs. In: Margaria, T., Yi, W. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 2031, pp. 268–283. Springer, Heidelberg (2001)
4. Ball, T., Rajamani, S.K.: The SLAM project: debugging system software via static analysis. In: Launchbury, J., Mitchell, J.C. (eds.) Symp. on Principles of Programming Languages (POPL), pp. 1–3. ACM, New York (2002)
5. Ball, T., Rajamani, S.K.: SLIC: a specification language for interface checking (of C). Tech. Rep. MSR-TR-2001-21, Microsoft Research (2002)
6. Beyer, D.: Status report on software verification (competition summary SV-COMP 2014). In: Ábrahám, E., Havelund, K. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 8413, pp. 373–388. Springer, Heidelberg (2014)
7. Beyer, D., Chlipala, A.J., Henzinger, T.A., Jhala, R., Majumdar, R.: Generating tests from counterexamples. In: Finkelstein, A., Estublier, J., Rosenblum, D.S. (eds.) Intl. Conf. on Software Engineering (ICSE), pp. 326–335. IEEE, Piscataway (2004)
8. Beyer, D., Chlipala, A.J., Henzinger, T.A., Jhala, R., Majumdar, R.: The BLAST query language for software verification. In: Giacobazzi, R. (ed.) Intl. Symp. on Static Analysis (SAS). LNCS, vol. 3148, pp. 2–18. Springer, Heidelberg (2004)
9. Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M.E., Sebastiani, R.: Software model checking via large-block encoding. In: Formal Methods in Computer Aided Design (FMCAD), pp. 25–32. IEEE, Piscataway (2009)
10. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M.: Correctness witnesses: exchanging verification results between verifiers. In: Intl. Symp. on Foundations of Software Engineering (FSE). ACM, New York (2016)
11. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Stahlbauer, A.: Witness validation and stepwise testification across software verifiers. In: Nitto, E.D., Harman, M., Heymans, P. (eds.) Intl. Symp. on Foundations of Software Engineering (FSE), pp. 721–733. ACM, New York (2015)
12. Beyer, D., Fararooy, A.: A simple and effective measure for complex low-level dependencies. In: Intl. Conf. on Program Comprehension (ICPC), pp. 80–83. IEEE, Piscataway (2010)
13. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker BLAST. Int. J. Softw. Tools Technol. Transf. **9**(5–6), 505–525 (2007)
14. Beyer, D., Henzinger, T.A., Majumdar, R., Rybalchenko, A.: Invariant synthesis for combined theories. In: Cook, B., Podelski, A. (eds.) Intl. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI). LNCS, vol. 4349, pp. 378–394. Springer, Heidelberg (2007)
15. Beyer, D., Henzinger, T.A., Majumdar, R., Rybalchenko, A.: Path invariants. In: Ferrante, J., McKinley, K.S. (eds.) Conf. on Programming Language Design and Implementation (PLDI), pp. 300–309. ACM, New York (2007)
16. Beyer, D., Henzinger, T.A., Théoduloz, G.: Lazy shape analysis. In: Ball, T., Jones, R.B. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 4144, pp. 532–546. Springer, Heidelberg (2006)
17. Beyer, D., Henzinger, T.A., Théoduloz, G.: Configurable software verification: concretizing the convergence of model checking and program analysis. In: Damm, W., Hermanns, H. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 4590, pp. 504–518. Springer, Heidelberg (2007)

18. Beyer, D., Henzinger, T.A., Théoduloz, G.: Program analysis with dynamic precision adjustment. In: Intl. Conf. on Automated Software Engineering (ASE), pp. 29–38. IEEE, Piscataway (2008)
19. Beyer, D., Henzinger, T.A., Théoduloz, G., Zufferey, D.: Shape refinement through explicit heap analysis. In: Rosenblum, D.S., Taentzer, G. (eds.) Intl. Conf. on Fundamental Approaches to Software Engineering (FASE). LNCS, vol. 6013, pp. 263–277. Springer, Heidelberg (2010)
20. Beyer, D., Holzer, A., Tautschnig, M., Veith, H.: Information reuse for multi-goal reachability analyses. In: Felleisen, M., Gardner, P. (eds.) European Symp. on Programming (ESOP). LNCS, vol. 7792, pp. 472–491. Springer, Heidelberg (2013)
21. Beyer, D., Keremoglu, M.E.: CPACHECKER: a tool for configurable software verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg (2011)
22. Beyer, D., Keremoglu, M.E., Wendler, P.: Predicate abstraction with adjustable-block encoding. In: Bloem, R., Sharygina, N. (eds.) Formal Methods in Computer Aided Design (FMCAD), pp. 189–197 (2010)
23. Beyer, D., Lemberger, T.: Symbolic execution with CEGAR. In: Intl. Conf. on Verified Software: Theories, Tools, and Experiments (VSTTE). LNCS. Springer, Heidelberg (2016)
24. Beyer, D., Löwe, S.: Explicit-state software model checking based on CEGAR and interpolation. In: Cortellessa, V., Varró, D. (eds.) Intl. Conf. on Fundamental Approaches to Software Engineering (FASE). LNCS, vol. 7793, pp. 146–162. Springer, Heidelberg (2013)
25. Beyer, D., Stahlbauer, A.: BDD-based software model checking with CPACHECKER. In: Kucera, A., Henzinger, T.A., Nesetril, J., Vojnar, T., Antos, D. (eds.) Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS). LNCS, vol. 7721, pp. 1–11. Springer, Heidelberg (2013)
26. Beyer, D., Wendler, P.: Reuse of verification results: conditional model checking, precision reuse, and verification witnesses. In: Bartocci, E., Ramakrishnan, C.R. (eds.) Intl. Symposium on Model Checking of Software (SPIN). LNCS, vol. 7976, pp. 1–17. Springer, Heidelberg (2013)
27. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, R. (ed.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
28. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: Conf. on Programming Language Design and Implementation (PLDI), pp. 196–207. ACM, New York (2003)
29. Bourdoncle, F.: Efficient chaotic iteration strategies with widenings. In: Bjørner, D., Broy, M., Pottosin, I.V. (eds.) Formal Methods in Programming and Their Applications. LNCS, vol. 735, pp. 128–141. Springer, Heidelberg (1993)
30. Brockschmidt, M., Cook, B., Fuhs, C.: Better termination proving through cooperation. In: Sharygina, N., Veith, H. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 8044, pp. 413–429. Springer, Heidelberg (2013)
31. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Trans. Comput. **C-35**(8), 677–691 (1986)
32. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. In: Symp. on Principles of Programming Languages (POPL), pp. 289–300. ACM, New York (2009)
33. Chaudhuri, S., Gulwani, S., Lublinerman, R.: Continuity analysis of programs. In: Hermenegildo, M.V., Palsberg, J. (eds.) Symp. on Principles of Programming Languages (POPL), pp. 57–70. ACM, New York (2010)
34. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM **50**(5), 752–794 (2003)

35. Clarke, E.M., Kröning, D., Sharygina, N., Yorav, K.: SATABS: SAT-based predicate abstraction for ANSI-C. In: Halbwachs, N., Zuck, L.D. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 3440, pp. 570–574. Springer, Heidelberg (2005)
36. Codish, M., Mulkers, A., Bruynooghe, M., de la Banda, M.G., Hermenegildo, M.: Improving abstract interpretations by combining domains. In: ACM Workshop on Partial Evaluation and Program Manipulation (PEPM), pp. 194–205. ACM, New York (1993)
37. Cormen, T.H., Leiserson, C.E., Rivest, R.L.: Introduction to Algorithms. MIT Press, Cambridge (1990)
38. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for the static analysis of programs by construction or approximation of fixpoints. In: Symp. on Principles of Programming Languages (POPL), pp. 238–252. ACM, New York (1977)
39. Cousot, P., Cousot, R.: Systematic design of program-analysis frameworks. In: Aho, A.V., Zilles, S.N., Rosen, B.K. (eds.) Symp. on Principles of Programming Languages (POPL), pp. 269–282. ACM, New York (1979)
40. Cousot, P., Cousot, R.: Formal language, grammar, and set-constraint-based program analysis by abstract interpretation. In: Intl. Conf. on Functional Programming Languages and Computer Architecture (FPCA), pp. 170–181. ACM, New York (1995)
41. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Symp. on Principles of Programming Languages (POPL), pp. 84–96. ACM, New York (1978)
42. Craig, W.: Linear reasoning. A new form of the Herbrand-Gentzen theorem. J. Symb. Log. **22**(3), 250–268 (1957)
43. Damas, L., Milner, R.: Principal type schemes for functional languages. In: Symp. on Principles of Programming Languages (POPL), pp. 207–212. ACM, New York (1982)
44. Dams, D., Namjoshi, K.S.: Orion: high-precision methods for static error analysis of C and C++ programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) Intl. Symp. on Formal Methods for Components and Objects (FMCO). LNCS, vol. 4111, pp. 138–160. Springer, Heidelberg (2005)
45. Dudka, K., Peringer, P., Vojnar, T.: Byte-precise verification of low-level list manipulation. In: Logozzo, F., Fähndrich, M. (eds.) Intl. Symp. on Static Analysis (SAS). LNCS, vol. 7935, pp. 215–237. Springer, Heidelberg (2013)
46. Dudka, K., Peringer, P., Vojnar, T.: Predator: a shape analyzer based on symbolic memory graphs (competition contribution). In: Ábrahám, E., Havelund, K. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 8413, pp. 412–414. Springer, Heidelberg (2014)
47. Ermis, E., Nutz, A., Dietsch, D., Hoenicke, J., Podelski, A.: Ultimate Kojak (competition contribution). In: Ábrahám, E., Havelund, K. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 8413, pp. 421–423. Springer, Heidelberg (2014)
48. Falke, S., Merz, F., Sinz, C.L.: Improved bounded model checking of C programs using LLVM (competition contribution). In: Piterman, N., Smolka, S.A. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 7795, pp. 623–626. Springer, Heidelberg (2013)
49. Fischer, J., Jhala, R., Majumdar, R.: Joining data flow with predicates. In: Intl. Symp. on Foundations of Software Engineering (FSE), pp. 227–236. ACM, New York (2005)
50. Geser, A., Knoop, J., Lüttgen, G., Rüthing, O., Steffen, B.: Non-monotone fixpoint iterations to resolve second-order effects. In: Gyimóthy, T. (ed.) Intl. Conf. on Compiler Construction (CC). LNCS, vol. 1060, pp. 106–120. Springer, Heidelberg (1996)
51. Giesl, J., Schneider-Kamp, P., Thiemann, R.: AProVE 1.2: automatic termination proofs in the dependency pair framework. In: Furbach, U., Shankar, N. (eds.) Intl. Joint Conf. on Automated Reasoning (IJCAR). LNAI, vol. 4130, pp. 281–286. Springer, Heidelberg (2006)

52. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
53. Gulwani, S., Jojic, N.: Program verification as probabilistic inference. In: Hofmann, M., Felleisen, M. (eds.) Symp. on Principles of Programming Languages (POPL), pp. 277–289. ACM, New York (2007)
54. Gulwani, S., Juvekar, S.: Bound analysis using backward symbolic execution. Tech. Rep. MSR-TR-2009-156, Microsoft Research (2009)
55. Gulwani, S., Lev-Ami, T., Sagiv, M.: A combination framework for tracking partition sizes. In: Shao, Z., Pierce, B.C. (eds.) Symp. on Principles of Programming Languages (POPL), pp. 239–251. ACM, New York (2009)
56. Gulwani, S., McCloskey, B., Tiwari, A.: Lifting abstract interpreters to quantified logical domains. In: Necula, G.C., Wadler, P. (eds.) Symp. on Principles of Programming Languages (POPL), pp. 235–246. ACM, New York (2008)
57. Gulwani, S., Necula, G.C.: A polynomial-time algorithm for global value numbering. In: Giacobazzi, R. (ed.) Intl. Symp. on Static Analysis (SAS). LNCS, vol. 3148, pp. 212–227. Springer, Heidelberg (2004)
58. Gulwani, S., Srivastava, S., Venkatesan, R.: Program analysis as constraint solving. In: Gupta, R., Amarasinghe, S.P. (eds.) Conf. on Programming Language Design and Implementation (PLDI), pp. 281–292. ACM, New York (2008)
59. Gulwani, S., Srivastava, S., Venkatesan, R.: Constraint-based invariant inference over predicate abstraction. In: Jones, N.D., Müller-Olm, M. (eds.) Intl. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI). LNCS, vol. 5403, pp. 120–135. Springer, Heidelberg (2009)
60. Gulwani, S., Tiwari, A.: Assertion checking over combined abstraction of linear arithmetic and uninterpreted functions. In: Sestoft, P. (ed.) European Symp. on Programming (ESOP). LNCS, vol. 3924, pp. 279–293. Springer, Heidelberg (2006)
61. Gulwani, S., Tiwari, A.: Combining abstract interpreters. In: Schwartzbach, M.I., Ball, T. (eds.) Conf. on Programming Language Design and Implementation (PLDI), pp. 376–386. ACM, New York (2006)
62. Gulwani, S., Tiwari, A.: Assertion checking unified. In: Cook, B., Podelski, A. (eds.) Intl. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI). LNCS, vol. 4349, pp. 363–377. Springer, Heidelberg (2007)
63. Gulwani, S., Tiwari, A.: Constraint-based approach for analysis of hybrid systems. In: Gupta, A., Malik, S. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 5123, pp. 190–203. Springer, Heidelberg (2008)
64. Gulwani, S., Zuleger, F.: The reachability-bound problem. In: Zorn, B.G., Aiken, A. (eds.) Conf. on Programming Language Design and Implementation (PLDI), pp. 292–304. ACM, New York (2010)
65. Gurfinkel, A., Albarghouthi, A., Chaki, S., Li, Y., Chechik, M.U.: Verification with interpolants and abstract interpretation (competition contribution). In: Piterman, N., Smolka, S.A. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 7795, pp. 637–640. Springer, Heidelberg (2013)
66. Gurfinkel, A., Belov, A.: FrankenBit: bit-precise verification with many bits (competition contribution). In: Ábrahám, E., Havelund, K. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 8413, pp. 408–411. Springer, Heidelberg (2014)
67. Heizmann, M., Christ, J., Dietsch, D., Hoenicke, J., Lindenmann, M., Musa, B., Schilling, C., Wissert, S., Podelski, A.: Ultimate Automizer with unsatisfiable cores (competition contribution). In: Ábrahám, E., Havelund, K. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 8413, pp. 418–420. Springer, Heidelberg (2014)

68. Heizmann, M., Hoenicke, J., Leike, J., Podelski, A.: Linear ranking for linear lasso programs. In: Hung, D.V., Ogawa, M. (eds.) Intl. Symp. Automated Technology for Verification and Analysis (ATVA). LNCS, vol. 8172, pp. 365–380. Springer, Heidelberg (2013)
69. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: Damm, W., Hermanns, H. (eds.) Symp. on Principles of Programming Languages (POPL), pp. 232–244. ACM, New York (2004)
70. Henzinger, T.A., Jhala, R., Majumdar, R., Necula, G.C., Sutre, G., Weimer, W.: Temporal-safety proofs for systems code. In: Brinksma, E., Larsen, K.G. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 2404, pp. 526–538. Springer, Heidelberg (2002)
71. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Launchbury, J., Mitchell, J.C. (eds.) Symp. on Principles of Programming Languages (POPL), pp. 58–70. ACM, New York (2002)
72. Inverso, O., Tomasco, E., Fischer, B., Torre, S.L., Parlato, G.: Lazy-CSeq: a lazy sequentialization tool for C (competition contribution). In: Ábrahám, E., Havelund, K. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 8413, pp. 398–401. Springer, Heidelberg (2014)
73. Johannes, K., Helmut, V.: JAKSTAB: a static analysis platform for binaries. In: Gupta, A., Malik, S. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 5123, pp. 423–427. Springer, Heidelberg (2008)
74. Jha, S., Gulwani, S., Seshia, S.A., Tiwari, A.: Oracle-guided component-based program synthesis. In: Kramer, J., Bishop, J., Devanbu, P.T., Uchitel, S. (eds.) Intl. Conf. on Software Engineering (ICSE), pp. 215–224. ACM, New York (2010)
75. Jones, N.D., Muchnick, S.S.: A flexible approach to interprocedural data-flow analysis and programs with recursive data structures. In: Symp. on Principles of Programming Languages (POPL), pp. 66–74. ACM, New York (1982)
76. Jung, Y., Kong, S., Wang, B.Y., Yi, K.: Deriving invariants by algorithmic learning, decision procedures, and predicate abstraction. In: Barthe, G., Hermenegildo, M.V. (eds.) Intl. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI). LNCS, vol. 5944, pp. 180–196. Springer, Heidelberg (2010)
77. Kam, J., Ullman, J.: Global data-flow analysis and iterative algorithms. J. ACM **23**(1), 158–171 (1976)
78. Kennedy, K.: A survey of data-flow analysis techniques. In: Jones, N.D., Muchniek, S.S. (eds.) Program Flow Analysis: Theory and Applications, pp. 5–54. Prentice Hall, New York (1981)
79. Klein, M., Knoop, J., Koschützki, D., Steffen, B.: DFA&OPT-METAFrame: a tool kit for program analysis and optimization. In: Margaria, T., Steffen, B. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 1055, pp. 422–426. Springer, Heidelberg (1996)
80. Knoop, J., Rüthing, O., Steffen, B.: Towards a tool kit for the automatic generation of interprocedural data-flow analyses. J. Program. Lang. **4**(4), 211–246 (1996)
81. Knoop, J., Rüthing, O., Steffen, B.: Lazy code motion. In: Conf. on Programming Language Design and Implementation (PLDI). ACM, New York (1992)
82. Knuth, D.E.: Semantics of context-free languages. Math. Syst. Theory **2**(2), 127–145 (1968)
83. Kong, S., Jung, Y., David, C., Wang, B.Y., Yi, K.: Automatically inferring quantified invariants via algorithmic learning from simple templates. In: Ueda, K. (ed.) Asian Symp. on Programming Languages and Systems (APLAS). LNCS, vol. 6461, pp. 328–343 (2010)
84. Kröning, D., Sharygina, N., Tsitovich, A., Wintersteiger, C.: Termination analysis with compositional transition invariants. In: Touili, T., Cook, B., Jackson, P. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 6174, pp. 89–103. Springer, Heidelberg (2010)
85. Kröning, D., Tautschnig, M.: CBMC: C bounded model checker (competition contribution). In: Ábrahám, E., Havelund, K. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 8413, pp. 389–391. Springer, Heidelberg (2014)

86. Lewis, P., Rosenkrantz, D., Stearns, R.: Compiler Design Theory. Addison-Wesley, Reading (1976)
87. Löwe, S., Mandrykin, M., Wendler, P.: CPAchecker with sequential combination of explicit-value analyses and predicate analyses (competition contribution). In: Ábrahám, E., Havelund, K. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 8413, pp. 392–394. Springer, Heidelberg (2014)
88. Mohnen, M.: A graph-free approach to data-flow analysis. In: Horspool, R.N. (ed.) Intl. Conf. on Compiler Construction (CC). LNCS, vol. 2304, pp. 46–61. Springer, Heidelberg (2002)
89. Morel, E., Renvoise, C.: Global optimization by suppression of partial redundancies. Commun. ACM **22**(1) (1979)
90. Morse, J., Ramalho, M., Cordeiro, L., Nicole, D., Fischer, B.: ESBMC 1.22 (competition contribution). In: Ábrahám, E., Havelund, K. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 8413, pp. 405–407. Springer, Heidelberg (2014)
91. Muller, P., Vojnar, T.: CPAlien: shape analyzer for CPAchecker (competition contribution). In: Ábrahám, E., Havelund, K. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 8413, pp. 395–397. Springer, Heidelberg (2014)
92. Müller-Olm, M., Seidl, H.: Precise interprocedural analysis through linear algebra. In: Jones, N.D., Leroy, X. (eds.) Symp. on Principles of Programming Languages (POPL), pp. 330–341. ACM, New York (2004)
93. Mycroft, A.: Polymorphic type schemes and recursive definitions. In: Paul, M., Robinet, B. (eds.) European Symp. on Programming (ESOP). LNCS, vol. 167, pp. 217–228. Springer, Heidelberg (1984)
94. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer, Heidelberg (1999)
95. Nordström, B., Petersson, K., Smith, J.: Programming in Martin-Löf's Type Theory. Oxford University Press, Oxford (1990)
96. Pasareanu, C.S., Dwyer, M.B., Visser, W.: Finding feasible counter-examples when model checking abstracted Java programs. In: Margaria, T., Yi, W. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 2031, pp. 284–298. Springer, Heidelberg (2001)
97. Pierce, B.C.: Types and Programming Languages. MIT Press, Cambridge (2002)
98. Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In: Steffen, B., Levi, G. (eds.) Intl. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI). LNCS, vol. 2937, pp. 239–251. Springer, Heidelberg (2004)
99. Popeea, C., Rybalchenko, A.: Threader: a verifier for multi-threaded programs (competition contribution). In: Piterman, N., Smolka, S.A. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 7795, pp. 633–636. Springer, Heidelberg (2013)
100. Reps, T.W., Horwitz, S., Sagiv, M.: Precise interprocedural data-flow analysis via graph reachability. In: Cytron, R.K., Lee, P. (eds.) Symp. on Principles of Programming Languages (POPL), pp. 49–61. ACM, New York (1995)
101. Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Global value numbers and redundant computations. In: Symp. on Principles of Programming Languages (POPL), pp. 12–27. ACM, New York (1988)
102. Rothermel, G., Harrold, M.: Analyzing regression test selection techniques. IEEE Trans. Softw. Eng. **22**(8), 529–551 (1996)
103. Sagiv, M., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. **24**(3), 217–298 (2002)
104. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Constraint-based linear-relations analysis. In: Giacobazzi, R. (ed.) Intl. Symp. on Static Analysis (SAS). LNCS, vol. 3148, pp. 53–68. Springer, Heidelberg (2004)

105. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Non-linear loop invariant generation using Gröbner bases. In: Symp. on Principles of Programming Languages (POPL), pp. 318–329. ACM, New York (2004)
106. Schmidt, D.A.: Denotational Semantics: A Methodology for Language Development. Allyn & Bacon, Needham Heights (1986)
107. Schmidt, D.A.: Data-flow analysis is model checking of abstract interpretations. In: Symp. on Principles of Programming Languages (POPL). ACM, New York (1998)
108. Schmidt, D.A., Steffen, B.: Program analysis as model checking of abstract interpretations. In: Levi, G. (ed.) Intl. Symp. on Static Analysis (SAS). LNCS, vol. 1503, pp. 351–380. Springer, Heidelberg (1998)
109. Shved, P., Mandrykin, M., Mutilin, V.: Predicate analysis with BLAST 2.7 (competition contribution). In: Flanagan, C., König, B. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 7214, pp. 525–527. Springer, Heidelberg (2012)
110. Slaby, J., Strejcek, J.: Symbiotic 2: more precise slicing (competition contribution). In: Ábrahám, E., Havelund, K. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 8413, pp. 415–417. Springer, Heidelberg (2014)
111. Srivastava, S., Gulwani, S.: Program verification using templates over predicate abstraction. In: Conf. on Programming Language Design and Implementation (PLDI), pp. 223–234. ACM, New York (2009)
112. Srivastava, S., Gulwani, S., Foster, J.S.: From program verification to program synthesis. In: Hermenegildo, M.V., Palsberg, J. (eds.) Symp. on Principles of Programming Languages (POPL), pp. 313–326. ACM, New York (2010)
113. Steffen, B.: Data-flow analysis as model checking. In: Ito, T., Meyer, A.R. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 536, pp. 346–365. Springer, Heidelberg (1991)
114. Steffen, B.: Generating data-flow analysis algorithms from modal specifications. Sci. Comput. Program. **21**(2), 115–139 (1993)
115. Steffen, B.: Property-oriented expansion. In: Cousot, R., Schmidt, D.A. (eds.) Intl. Symp. on Static Analysis (SAS). LNCS, vol. 1145, pp. 22–41. Springer, Heidelberg (1996)
116. Steffen, B., Claßen, A., Klein, M., Knoop, J., Margaria, T.: The fixpoint-analysis machine. In: Lee, I., Smolka, S.A. (eds.) Intl. Conf. on Concurrency Theory (CONCUR). LNCS, vol. 962, pp. 72–87. Springer, Heidelberg (1995)
117. Tomasco, E., Inverso, O., Fischer, B., Torre, S.L., Parlato, G.: MU-CSeq: sequentialization of C programs by shared memory unwindings (competition contribution). In: Ábrahám, E., Havelund, K. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 8413, pp. 402–404. Springer, Heidelberg (2014)
118. Urban, C., Miné, A.: An abstract domain to infer ordinal-valued ranking functions. In: Shao, Z. (ed.) European Symp. on Programming (ESOP). LNCS, vol. 8410, pp. 412–431. Springer, Heidelberg (2014)
119. Vizel, Y., Grumberg, O., Shoham, S.: Intertwined forward-backward reachability analysis using interpolants. In: Piterman, N., Smolka, S.A. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 7795, pp. 308–323. Springer, Heidelberg (2013)
120. Šerý, O.: Enhanced property specification and verification in BLAST. In: Chechik, M., Wirsing, M. (eds.) Intl. Conf. on Fundamental Approaches to Software Engineering (FASE). LNCS, vol. 5503, pp. 456–469. Springer, Heidelberg (2009)

Chapter 17
Model Checking Procedural Programs

Rajeev Alur, Ahmed Bouajjani, and Javier Esparza

Abstract We consider the model-checking problem for sequential programs with procedure calls. We first present basic algorithms for solving the reachability problem and the fair computation problem. The algorithms are based on two techniques: *summarization*, which computes reachability information by solving a set of fixpoint equations, and *saturation*, which computes the set of all reachable program states (including call stacks) using automata. Then, we study formalisms to specify requirements of programs with procedure calls. We present an extension of Linear Temporal Logic allowing propagation of information across the hierarchical structure induced by procedure calls and matching returns. Finally, we show how model checking can be extended to this class of programs and properties.

17.1 Introduction

We consider the model-checking problem for sequential programs consisting of procedures that call one another, possibly in a recursive manner. We assume that all program variables have a finite range. These programs, called procedural programs or Boolean programs [9], are used as abstractions of C programs in highly influential software verification tools like SLAM [8]. The state of a procedural program has three parts: the current value of the program counter, the current values of the program variables, and the current stack of procedure calls whose execution has not yet finished. Since procedures may be recursive, and the recursion depth is not bounded a priori, the state space of a procedural program may be infinite, and so procedural programs cannot be verified using standard finite-state model-checking algorithms.

R. Alur
University of Pennsylvania, Philadelphia, PA, USA

A. Bouajjani
Institut Universitaire de France, Paris Diderot University (Paris 7), Paris, France

J. Esparza (✉)
Technical University of Munich, Munich, Germany
e-mail: esparza@in.tum.de

E.M. Clarke et al. (eds.), *Handbook of Model Checking*,
DOI 10.1007/978-3-319-10575-8_17

We model procedural programs as recursive state machines (RSMs) [2]. Each procedure of the program is modeled by a different machine. A machine has a finite number of control states with some distinguished entry and exit points. Control states are connected by edges that correspond to either a local change in the control state, or to a full execution (from an entry to an exit point) of another state machine. The latter case models a procedure call. Recursion is allowed since the dependencies among state machines can be cyclic in general. RSMs with acyclic dependencies among state machines are called hierarchical state machines [7].

The operational semantics of RSMs can be defined in terms of pushdown systems (PDSs), state machines whose transitions are labeled with stack operations [12, 29]. Push and pop operations correspond to procedure calls and returns, respectively. The PDS corresponding to a given RSM can be easily computed and has roughly the same size as the RSM itself, and either representation can serve as an input to a verification algorithm, depending on the computational task at hand.

We present two basic techniques for the reachability analysis of RSMs and PDSs. The first one, called summarization, computes reachability information by solving a set of fixpoint equations, and is closely related to inter-procedural data-flow analysis [25, 49]. Roughly speaking, summarization computes the pairs of program counter and program variable values that can be reached from the initial state of the program, but not the stack contents with which they can be reached. For example, after applying summarization we may know that program location 13 can be reached with $x = 3$ and $y = 5$, but not that this can only happen when the current procedure is called from a procedure P. The complete set of reachable program states (i.e., the set of all reachable triples consisting of the current value of the program counter, the current values of the program variables, and the current stack of procedure calls) can be obtained by employing the second technique, called saturation. Saturation takes as input the PDS associated with the RSM, and computes its set of reachable configurations. Since this set may be infinite, saturation does not enumerate its elements, but computes a finite symbolic representation in the shape of a finite automaton (compare with the BDD-based techniques of Chap. 8 [22] for compactly representing a large but finite set of states). Summarization and saturation can also be applied to the fair computation problem, a core computational problem underlying the analysis of infinite program executions. They have been implemented and extensively applied in tools like Bebop (the model checker inside SLAM) [9], MOPED and jMOPED [28, 47, 52], and WALi [37].

In the second part of the chapter we discuss extensions of automata and logics suitable for specifying properties of procedural programs. We show that many natural specifications require relating the truth of propositions at a procedure call with the matching return position. A typical example is the property "if the status of a global variable x is *locked* when a procedure P is called, then its status is guaranteed to be *locked* when the procedure P returns". Asserting such properties is not possible using formalisms defining regular languages of computations, such as finite-state automata and Linear Temporal Logic (see Chap. 2 [42]). Aimed at specifying such properties, the notion of nested words is introduced to represent behaviors of RSMs. They correspond to (finite/infinite) words with additional hierarchical edges that expose the matching between call and return positions. We define

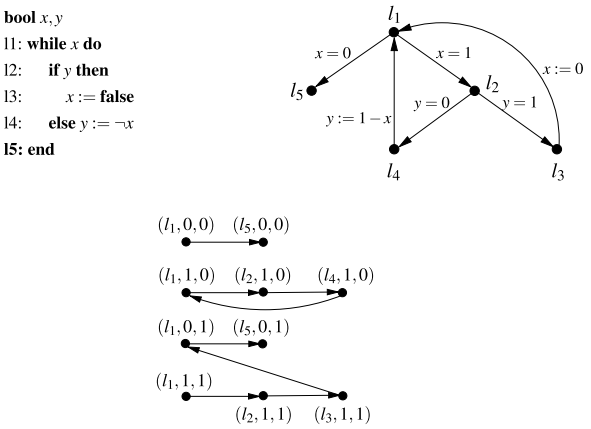


Fig. 1 A program, its extended state machine, and its state machine

automata and logics on nested words, and show that model-checking algorithms for RSMs naturally extend to these formalisms.

In the last section of the chapter, we survey some further existing results concerning RSMs/PDSs and their extensions that are relevant to the domain of verification.

17.2 Models of Procedural Programs

While programs, consisting of assignment statements, if-then-else statements and while loops, can be modeled as *extended state machines*: state machines whose transitions are guarded by and operate on variables. The states or *nodes* of an extended state machine correspond to the control points of the program. The transitions of the machine are labeled either with assignments or with the Boolean conditions appearing in the conditional statements and the while loops. Figure 1 shows an example of a while program with two boolean variables x, y (top left), and its corresponding extended state machine (top right). The nodes l_1 and l_5 are called the *entry* and *exit* nodes, respectively.

An extended state machine with a set V of variables can be *flattened* into a *state machine*. A node of the state machine is a pair $\langle \ell, v \rangle$, where ℓ is a node of the extended state machine, and v is a valuation of the variables of V. Figure 1 shows at the bottom the state machine obtained by flattening the extended state machine. For instance, the entry node l_1 is split into four entry nodes, one for each possible valuation of the variables x and y. Only the nodes of the state machine reachable from the entry nodes are shown.

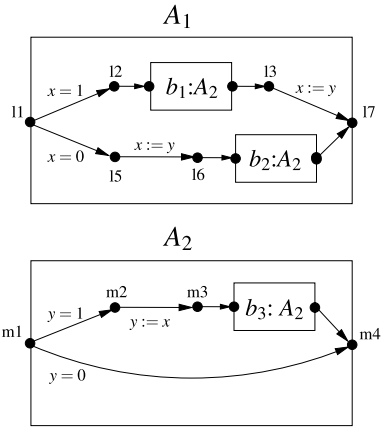
Procedural programs extend while programs with (possibly recursive) procedures. They can no longer be faithfully modeled by state machines. For this reason we introduce two abstract models of computation, *recursive state machines*, and *pushdown systems*, which play for procedural programs the same role that state machines play for while programs.

Fig. 2 A procedural program and its extended recursive state machine

```
bool x, y

proc P1
l1: if x then
l2:     call P2 do
l3:     x := y
l4: else
l5:     x := y
l6:     call P2 do
l7: end

proc P2
m1: if y then
m2:     y := x
m3:     call P2 do
m4: return
```



17.2.1 (Extended) Recursive State Machines

Figure 2 shows a procedural program and its corresponding extended recursive state machine (ERSM). The program has two global Boolean variables x and y, and consists of two procedures P_1 and P_2. The ERSM reflects the structure of the program. It consists of two *components* A_1 and A_2, modeling the procedures P_1 and P_2, respectively. The nodes of A_1 and A_2 correspond to the control points of P_1 and P_2; assignments, conditionals, etc. are modeled as for while programs. Moreover, for each call in procedure P_i to the procedure P_j, the component A_i contains a *box* labeled by A_j. In our example, component A_1 contains two boxes, and component A_2 contains one box. Each box has an *entry port* and an *exit port*. Ports are pairs (n, b), where b is a box, n is an entry or exit node of $A_{Y(b)}$, and $Y(b)$ denotes the component called by the box b. A transition leads from the control point at which the call is made to the entry port, and a second transition leads from the exit port to the return address (the control point at which the computation of the caller continues after the execution of the callee returns).

As in the case of extended state machines, ERSMs can be flattened into *recursive state machines*. Flattening preserves the number of components and boxes, but multiplies the number of nodes and ports. As for state machines, a node or a port of a recursive state machine is a pair $\langle \ell, v \rangle$, where ℓ is a state of the extended machine, and v is a valuation of the variables. Figure 3 shows the result of flattening component A_2 of Fig. 2.

Definition 1 A *recursive state machine* (RSM) is a tuple $\mathcal{M} = (A_1, \ldots, A_k)$ of *components* $A_i = (N_i, B_i, Y_i, En_i, Ex_i, \delta_i)$, where:

- N_i is a finite set of *nodes*, with two distinguished subsets En_i and Ex_i of *entry* and *exit* nodes.
- B_i is a finite set of *boxes*. A box b is labeled with an integer $Y(b) \in \{1, \ldots, k\}$, and has a *call port* (en, b) for each entry node en of $A_{Y(b)}$, and a *return port* (ex, b) for each exit node ex of $A_{Y(b)}$.

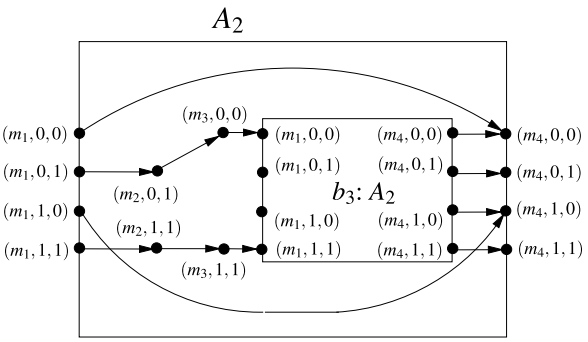


Fig. 3 Result of flattening component A_2 of Fig. 2

- δ_i is a set of *transitions* $u \to v$ where u is either a non-exit node or a return port, and v is either a non-entry node, or a call port.

We denote by

$$En = \bigcup_{i=1}^{k} En_i \qquad Ex = \bigcup_{i=1}^{k} Ex_i \qquad N = \bigcup_{i=1}^{k} N_i \qquad B = \bigcup_{i=1}^{k} B_i$$

the set of all entry nodes, exit nodes, nodes, and boxes of $\mathcal{M}$, respectively. The set of all ports is $\Pi = (En \cup Ex) \times B$.

Observe that there are four kinds of transitions: $n \to m$ (node-to-node), $n \to (en, b)$ (node-to-call-port), $(ex, b) \to m$ (return-port-to-node), and $(ex, b) \to (en, b')$ (return-port-to-call-port).

In the component of Fig. 3, the entry and exit nodes are the triples with m_1 and m_4 as first element, respectively. The only box is b_3, and $Y(b_3) = 2$. The call ports are $((m_1, 0, 0), b3), \ldots, ((m_1, 1, 1), b_3)$. In the figure the ports are labeled just by $(m_1, 0, 0), \ldots, (m_1, 1, 1)$, and the return ports are $((m_4, 0, 0), b_3), \ldots, ((m_4, 1, 1), b_3)$.

17.2.2 Pushdown Systems

Intuitively, an RSM can be executed using a *stack* that at every point in the computation contains the sequence of boxes that have been entered but not yet exited. If a component enters a box b (which corresponds to calling the procedure modeled by the component $A_{Y(b)}$), then b is pushed onto the stack; if the component exits the box b (which corresponds to a return from the called procedure), then b is popped. This suggests an operational semantics for RSMs in terms of pushdown systems.

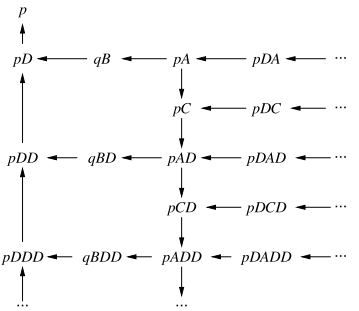
Definition 2 A *pushdown system* (PDS) is a triple $\mathcal{P} = (P, \Gamma, \Delta)$, where P is a finite set of *control states*, Γ is a finite *stack alphabet*, and Δ is a finite set of *rules* of the form $pX \hookrightarrow q\alpha$ with $p, q \in P$, $X \in \Gamma \cup \{\epsilon\}$, and $\alpha \in \Gamma^*$.

Fig. 4 A PDS and its transition system

$P = \{p, q\}$

$\Gamma = \{A, B, C, D\}$

$\Delta = \{\ pA \hookrightarrow qB,\ pA \hookrightarrow pC,\ qB \hookrightarrow pD,\ pC \hookrightarrow pAD,\ pD \hookrightarrow p\ \}$



A *configuration* of a PDS is a string of the form $p\sigma$, where $p \in P$ and $\sigma \in \Gamma^*$. The transition system associated with a PDS is the graph having the configurations as vertices, and an edge $c \to c'$ between two configurations c and c' if there is a rule $pX \hookrightarrow q\alpha$ and a word $\sigma \in \Gamma^*$ such that $c = pX\sigma$ and $c' = q\alpha\sigma$. We then say that c is an *immediate predecessor* of c' and c' an *immediate successor* of c.

Figure 4 shows a PDS and a fragment of its transition system. Notice that the transition system of a PDS may be infinite, even if we only consider the configurations reachable from some initial configuration.

17.2.3 From RSMs to PDSs

Loosely speaking, the PDS associated with an RSM is the pushdown machine that executes the RSM. In programming terms, an RSM is a formal model of a procedural program, and its corresponding PDS is a formal model of the executable code of the program.

Formally, the PDS $\mathcal{P}_\mathcal{M} = (P_\mathcal{M}, \Gamma_\mathcal{M}, \Delta_\mathcal{M})$ corresponding to an RSM $\mathcal{M} = (A_1, \ldots, A_k)$ is defined as follows:

- $P_\mathcal{M} = N$ is the set of all nodes of $\mathcal{M}$;
- $\Gamma_\mathcal{M} = B$ is the set of all boxes of $\mathcal{M}$; and
- $\Delta_\mathcal{M}$ is the set containing
  - a rule $n \hookrightarrow m$ for each transition $n \to m$;
  - a rule $n \hookrightarrow en\, b$ for every transition $n \to (en, b)$;
  - a rule $ex\, b \hookrightarrow m$ for every transition $(ex, b) \to m$; and
  - a rule $ex\, b \hookrightarrow en\, b'$ for every transition $(ex, b) \to (en, b')$.

Observe that the PDS has exactly one rule for each transition of the RSM.

As an example, for the RSM obtained by flattening the ERSM of Fig. 2, we get $P_\mathcal{M} = \{l_1, \ldots, l_7, m_1, \ldots, m_4\} \times \{0, 1\} \times \{0, 1\}$ and $\Gamma_\mathcal{M} = \{b_1, b_2, b_3\}$. Examples of rules of $\Delta_\mathcal{M}$ are

$$\begin{array}{rcll} (m_1, 0, 0) & \hookrightarrow & (m_4, 0, 0) & \text{derived from } (m_1, 0, 0) \to (m_4, 0, 0) \\ (m_3, 0, 0) & \hookrightarrow & (m_1, 0, 0)\, b_3 & \text{derived from } (m_3, 0, 0) \to ((m_1, 0, 0), b_3) \\ (m_4, 0, 1)\, b_3 & \hookrightarrow & (m_4, 0, 1) & \text{derived from } ((m_4, 0, 1), b_3) \to (m_4, 0, 1) \end{array}$$

Observe that every rule $pX \hookrightarrow q\alpha$ of a PDS associated with an RSM satisfies $|\alpha| \leq 2$.

17.3 Basic Verification Algorithms

We proceed to define basic computational problems that are useful for checking safety and liveness properties of RSMs.

Definition 3 Let $\mathcal{M} = (A_1, \ldots, A_k)$ be an RSM, where $A_i = (N_i, B_i, Y_i, En_i, Ex_i, \delta_i)$ for each $i \in \{1, \ldots, k\}$. Let $\xrightarrow{*}$ be the reflexive-transitive closure of the relation $\rightarrow$ between configurations, i.e., $\xrightarrow{*} = \bigcup_{n=0}^{\infty} (\rightarrow)^n$ and let $\xrightarrow{+} = \bigcup_{n=1}^{\infty} (\rightarrow)^n$.

The *state reachability problem* is to determine, given an entry node $p \in En$ and a node $q \in P_{\mathcal{M}}$, whether $p \xrightarrow{*} q\sigma$ for some $\sigma \in \Gamma_{\mathcal{M}}^*$.

The *configuration reachability problem* is to determine, given two configurations $p\sigma$ and $p'\sigma'$, where $p, p' \in P_{\mathcal{M}}$ and $\sigma, \sigma' \in \Gamma_{\mathcal{M}}^*$, whether $p\sigma \xrightarrow{*} p'\sigma'$.

The *fair computation problem* is to determine, given an entry node $p \in En$ and a finite set of *repeat* entry nodes $F \subseteq En$, whether p has an F-fair computation, i.e., an infinite sequence of configurations $p_0\sigma_0, p_1\sigma_1, p_2\sigma_2, \ldots$ such that (1) $p_0 = p$ and $\sigma_0 = \epsilon$, (2) $p_i\sigma_i \xrightarrow{+} p_{i+1}\sigma_{i+1}$ for every $i \geq 0$, and (3) $p_j \in F$ for infinitely many $j \geq 0$.

Consider the RSM of Fig. 3 on its own (not as part of the larger RSM obtained by flattening the extended RSM of Fig. 2). Choose p as the entry node $(m_1, 0, 1)$, and q as the node $(m_4, 1, 0)$. The state reachability problem for this choice of p and q formalizes the question whether some computation of procedure P_2 of Fig. 2 with $x = 0$ and $y = 1$ can reach the point m_4 with $x = 1$ and $y = 0$. However, since the procedure P_2 is recursive, m_4 can be visited several times during a computation, and so the question is whether at one of these visits x and y are equal to 1 and 0, respectively, not whether these are the values *after termination*. To check this we can use the configuration reachability problem: Procedure P_2 terminates with $x = 1$ and $y = 0$ if and only if the RSM can reach the configuration with $(m_4, 1, 0)$ as control state and empty stack. Notice that, in general, we cannot reduce termination (a liveness property) to reachability (a safety property), but inspection of this program shows that it terminates if and only if it reaches m_4 with no pending procedure calls.

The problem of checking liveness properties can be easily reduced to the *fair computation problem* by means of the automata-theoretic techniques introduced in Chap. 4 [38].

17.3.1 The State Reachability Problem: Computing Summaries

In this section we show how to solve the state reachability problem using the *summarization technique*. We present the technique for RSMs.

Let $\mathcal{M} = (A_1, \ldots, A_k)$ be an RSM, and let $\Theta_{\mathcal{M}} = N \cup \Pi$ be the set containing all nodes and all ports in $\mathcal{M}$. For every $i \in \{1, \ldots, k\}$, consider the relation $R_i \subseteq \Theta_{\mathcal{M}} \times \Theta_{\mathcal{M}}$ given by

$$(p, q) \in R_i \text{ iff } p \in En_i,\ q \text{ is a node or port of } A_i, \text{ and } p \xrightarrow{*} q.$$

Further, for every $i, j \in \{1, \ldots, k\}$ consider the relation $R_{(i,j)} \subseteq \Theta_{\mathcal{M}} \times \Theta_{\mathcal{M}}$ given by

$$(p, q) \in R_{(i,j)} \text{ iff } p \in En_i,\ q \text{ is a node or port of } A_j, \text{ and } p \xrightarrow{*} q\sigma \text{ for some } \sigma \in \Gamma^*_{\mathcal{M}}.$$

We call these relations *summaries*, since they can be seen as the result of summarizing executions by their initial and final states. Now, let $R = \bigcup_{i,j=1}^{k} R_{(i,j)}$. Clearly, given $p \in En$ and $q \in N$, solving the state reachability problem consists of checking whether $(p, q) \in R$.

It is easy to see that for every $i, j \in \{1, \ldots, k\}$ the relations R_i and $R_{(i,j)}$ are the smallest relations satisfying the following conditions (where we write $R_i(p, q)$ and $R_{(i,j)}(p, q)$ instead of $(p, q) \in R_i$ and $(p, q) \in R_{(i,j)}$):

S1: $R_i(e, e)$ for every $e \in En_i$.

S2: If $R_i(e, p)$ and $(p, q) \in \delta_i$, where $e \in En_i$, then $R_i(e, q)$.

S3: If $R_i(e, (p, b))$ and $R_\ell(p, q)$, where $e \in En_i$, $Y_i(b) = \ell$, $p \in En_\ell$, and $q \in Ex_\ell$, then $R_i(e, (q, b))$.

S4: If $R_i(e, q)$ then $R_{(i,i)}(e, q)$.

S5: If $R_i(e, (p, b))$ and $R_{(\ell,j)}(p, q)$, where $e \in En_i$, $Y_i(b) = \ell$, $p \in En_\ell$, and $q \in N_j$, then $R_{(i,j)}(e, q)$.

The relations R_i and $R_{(i,j)}$ can be simultaneously computed by, starting from the empty relations, iteratively applying the rules **S1**-**S5** until stabilization. Since the set $\Theta_{\mathcal{M}}$ is finite, the computation necessarily terminates. This yields a decision procedure of polynomial complexity for the state reachability problem. More precisely, as shown in [2], reachability can be solved in time $O(|\mathcal{M}|\theta_e^2)$ and space $O(|\mathcal{M}|\theta_e)$, where $|\mathcal{M}|$ is the total number of nodes and transitions in the RSM, and θ_e is the maximum number of entry nodes of a component, i.e., $\theta_e = \max_{i=1}^{k} |En_i|$.

It is straightforward to define a dual algorithm that starts at the exit nodes and computes the summaries *backwards*. For instance, in the dual algorithm the rule **S2** is replaced by the dual rule

D2: If $R_i(p, x)$ and $(q, p) \in \delta_i$, where $x \in Ex_i$, then $R_i(q, x)$.

The dual algorithm runs in $O(|\mathcal{M}|\theta_x^2)$ time, where θ_x is the maximum number of exit nodes of a component, i.e., $\theta_x = \max_{i=1}^{k} |Ex_i|$. The primal and dual rules can

be combined component-wise: if the number of entry nodes of component A_i is smaller than its number of exit nodes, then we compute R_i from the entry nodes using the primal rules, otherwise from the exit nodes using the dual rules. The complexity of this algorithm is $O(|\mathcal{M}|\theta^2)$, where $\theta = max_{i=1}^{k} min(|En_i|, |Ex_i|)$. Since $\theta \in O(|\mathcal{M}|)$, the combined algorithm has cubic complexity in $|\mathcal{M}|$. For the class of RSMs in which θ is bounded by a constant (which contains in particular procedural programs whose procedures can only return a fixed number of values, say a Boolean), reachability can be decided in linear time.

As an example, we compute part of the relations for the RSMs obtained by flattening the extended machines of Fig. 2. In particular, we show that

$$R_1\big((\ell_1, 1, 0), (\ell_7, 0, 0)\big)$$

holds, i.e., if we start at location ℓ_1 with $x = 1$ and $y = 0$, we can reach location ℓ_7 with $x = 0 = y$.

We first apply rule (**S1**) twice and obtain

$$R_1\big((\ell_1, 1, 0), (\ell_1, 1, 0)\big) \tag{1}$$

$$R_2\big((m_1, 1, 0), (m_1, 1, 0)\big) \tag{2}$$

Now we use rule (**S2**) to establish relations corresponding to single edges in the graphs of the RSMs. From (1), $((\ell_1, 1, 0), (\ell_2, 1, 0)) \in \delta_1$ and $((\ell_2, 1, 0), ((m_1, 1, 0), b_1)) \in \delta_1$, and from (2) and $((m_1, 1, 0), (m_4, 1, 0)) \in \delta_2$, respectively, we obtain

$$R_1\big((\ell_1, 1, 0), \big((m_1, 1, 0), b_1\big)\big) \tag{3}$$

$$R_2\big((m_1, 1, 0), (m_4, 1, 0)\big) \tag{4}$$

Next we apply rule (**S3**) to (3) and (4). Together with $Y_1(b_1) = 2$ we get

$$R_1\big((\ell_1, 1, 0), \big((m_4, 1, 0), b_1\big)\big) \tag{5}$$

Then we apply rule (**S2**) to (5) using the transition $(((m_4, 1, 0), b_1), (\ell_3, 1, 0)) \in \delta_1$, to obtain

$$R_1\big((\ell_1, 1, 0), (\ell_3, 1, 0)\big) \tag{6}$$

Finally, applying rule (**S2**) to (6) and $((\ell_3, 1, 0), (\ell_7, 0, 0)) \in \delta_1$ yields

$$R_1\big((\ell_1, 1, 0), (\ell_7, 0, 0)\big) \tag{7}$$

and we are done.

Let us now show

$$R_{(1,2)}\big((\ell_1, 0, 1), (m_3, 1, 1)\big)$$

Applying rule (**S1**) and then (**S2**) to the entry nodes $(\ell_1, 0, 1)$ and $(m_1, 1, 1)$ we obtain

$$R_1\big((\ell_1, 0, 1), \big((m_1, 1, 1), b_2\big)\big) \tag{8}$$

$$R_2\big((m_1, 1, 1), (m_3, 1, 1)\big) \tag{9}$$

Applying rule (**S4**) to (9) yields

$$R_{(2,2)}\big((m_1, 1, 1), (m_3, 1, 1)\big) \tag{10}$$

Finally, applying rule (**S5**) to (8) and (10) we get

$$R_{(1,2)}\big((\ell_1, 0, 1), (m_3, 1, 1)\big) \tag{11}$$

Next we show that calls to A_2 starting from $(m_1, 1, 1)$ never return, i.e., that $R_2((m_1, 1, 1), n)$ does not hold for any exit node n of A_2. Since every path from the entry node $(m_1, 1, 1)$ leads to $(m_2, 1, 1)$ and $(m_3, 1, 1)$, rule ($\mathbf{S_2}$) only allows us to derive $R_2((m_1, 1, 1), (m_2, 1, 1))$, $R_2((m_1, 1, 1), (m_3, 1, 1))$, and $R_2((m_1, 1, 1), ((m_1, 1, 1), b_3))$. Since no other rule can be applied, we are done.

Finally, similar reasoning shows that no exit node of A_1 is reachable from $(\ell_1, 0, 1)$. Indeed, this follows easily from the fact that rule (**S3**) cannot be applied to (8).

17.3.2 The Fair Computation Problem

It is shown in [12] that the fair computation problem can be reduced to the state reachability problem. The key observation, not difficult to prove, is that, given $p \in En$ and $F \subseteq En$, the node p has an F-fair computation if and only if there exists $p' \in F$ such that

$$p \xrightarrow{*} p'\sigma \text{ for some } \sigma \in \Gamma^*_{\mathcal{M}} \quad \text{and} \quad p' \xrightarrow{+} p'\sigma' \text{ for some } \sigma' \in \Gamma^*_{\mathcal{M}}.$$

This reduction allows us to solve the fair computation problem using summarization. We define a new reachability relation $R' \subseteq \Theta_{\mathcal{M}} \times \Theta_{\mathcal{M}}$ (in addition to the relation R defined in Sect. 17.3.1) as follows:

$$R'(p, q) \text{ holds if and only if } p \xrightarrow{+} q\sigma \text{ for some } \sigma \in \Gamma^*_{\mathcal{M}}.$$

Then, by the observation above, p has an F-fair computation if and only if there exists $p' \in F$ such that $R(p, p')$ and $R'(p', p')$.

The relation R' can be computed similarly to the relation R in Sect. 17.3.1. For every $i \in \{1, \ldots, k\}$, define $R'_i \subseteq \Theta_{\mathcal{M}} \times \Theta_{\mathcal{M}}$ by

$$R'_i(p, q) \text{ iff } p \in En_i,\ q \text{ is a node or port of } A_i, \text{ and } p \xrightarrow{+} q.$$

Further, for every $i, j \in \{1, \ldots, k\}$, define $R'_{(i,j)} \subseteq \Theta_{\mathcal{M}} \times \Theta_{\mathcal{M}}$ by

$$R'_{(i,j)}(p, q) \text{ iff } p \in En_i,\ q \text{ is a node or port of } A_j, \text{ and } p \xrightarrow{+} q\sigma \text{ for some } \sigma \in \Gamma^*_{\mathcal{M}}.$$

We clearly have $R' = \bigcup_{i,j=1}^{k} R'_{(i,j)}$.

For all $i, j \in \{1, \ldots, k\}$, the relations R'_i and $R'_{(i,j)}$ are the smallest relations such that:

S2': If $R_i(e, p)$ or $R'_i(e, p)$, and $(p, q) \in \delta_i$, where $e \in En_i$, then $R'_i(e, q)$.

S3': If $R'_i(e, (p, b))$ and $R_\ell(p, q)$, where $e \in En_i$, $Y_j(b) = \ell$, $p \in En_\ell$, and $q \in Ex_\ell$, then $R'_i(e, (q, b))$.

S4': If $R'_i(e, q)$, where $e \in En_i$, then $R'_{(i,i)}(e, q)$.

S5': If $R'_i(e, (p, b))$ and $R_{(\ell,j)}(p, q)$, where $e \in En_i$, $Y_i(b) = \ell$, $p \in En_\ell$, and $q \in N_j$, then $R'_{(i,j)}(e, q)$.

The relations can again be computed by applying the rules until stabilization. The time complexity is again cubic in $|\mathcal{M}|$, and linear if each component has a small number of either enter or exit nodes [2].

The model-checking problem for Linear Temporal Logic can be reduced to the fair computation problem using the automata-theoretic techniques of Chap. 4 [38] and Sect. 17.4.

17.3.3 The Configuration Reachability Problem: Saturating Automata

In this section we solve the configuration reachability problem for RSMs and PDSs. We present two decision procedures for PDSs. The procedures for RSMs are obtained by applying the translation from RSMs to PDSs shown in Sect. 17.2.3.

Given two configurations $p\sigma$ and $p'\sigma'$ of a PDS, we can decide whether $p\sigma \xrightarrow{*} p'\sigma'$ holds by computing the set of all configurations reachable from $p\sigma$ and checking whether $p'\sigma'$ belongs to it, or by computing the set of all configurations from which $p'\sigma'$ can be reached and checking whether $p\sigma$ belongs to it. Since these sets may be infinite, we have to explain the meaning of "compute". A configuration $p\sigma$ of a PDS can be seen as a word over the union of the set of control states and stack symbols, and so a set of configurations is a language over the same alphabet. Recall that a language is regular if it is recognized by a finite automaton. It turns out that, given a regular set C of configurations, the set of configurations reachable from C and the set of configurations from which C can be reached are again regular. This theorem, which can be traced back to Büchi (see Chap. 5 of [15]), allows us to define "computing the set" as "computing a finite automaton recognizing the set".

We fix a PDS $\mathcal{P} = (P, \Gamma, \Delta)$ for the rest of the section, and let $\mathcal{C}$ denote the set of all configurations of $\mathcal{P}$. The successor function $post: 2^{\mathcal{C}} \to 2^{\mathcal{C}}$ of $\mathcal{P}$ is defined as follows: c belongs to $post(C)$ if some immediate predecessor of c belongs to C. The reflexive and transitive closure of $post$ is denoted by $post^*$ and so, given a set

C of configurations, $post^*(C)$ denotes the set of configurations reachable from C. Similarly, we define $pre(C)$ as the set of immediate predecessors of elements in C and pre^* as the reflexive and transitive closure of pre.

It is convenient to define a variant of finite automata tailored for the task of representing sets of configurations of $\mathcal{P}$. A *$\mathcal{P}$-automaton* is an automaton with Γ as its alphabet, and P as the set of initial states. Formally, a $\mathcal{P}$-automaton is an automaton $\mathcal{A} = (\Gamma, Q, \delta, P, F)$ where Q is the finite set of states, $\delta \subseteq Q \times \Gamma \times Q$ is the set of *transitions*, $P \subseteq Q$ is the set of *initial states* and $F \subseteq Q$ the set of *final states*.

All the automata used in this section are $\mathcal{P}$-automata, so we drop the $\mathcal{P}$ from now on. An automaton *accepts* or *recognizes* a configuration $p\sigma$ if $p \xrightarrow{\sigma} q$ for some $q \in F$, where $p \xrightarrow{\sigma} q$ denotes that there is a path from state p to state q labeled by σ. A set of configurations of $\mathcal{P}$ is *regular* if it is recognized by some automaton.

In the next sections we present algorithms that given an automaton recognizing a set C of configurations compute automata recognizing $post^*(C)$ and $pre^*(C)$. We start with $pre^*(C)$, since in this case the algorithm is a bit simpler.

17.3.3.1 Computing $pre^*(C)$ for a Regular Set C by Saturation

The input to our algorithm is an automaton $\mathcal{A}$ accepting C. Without loss of generality, we assume that $\mathcal{A}$ has no transitions leading to an initial state (by adding new initial states if necessary, every automaton can be easily transformed into another one satisfying this condition and recognizing the same language). We compute $pre^*(C)$ as the language accepted by an automaton $\mathcal{A}_{pre^*}$ obtained from $\mathcal{A}$ by means of a *saturation procedure*. The procedure adds new transitions to $\mathcal{A}$, but no new states. New transitions are added according to the following *saturation rule*:

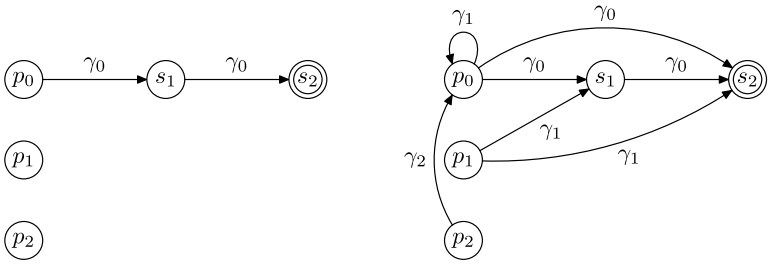
> If $p\gamma \hookrightarrow p'\sigma$ and $p' \xrightarrow{\sigma} q$ in the current automaton, then add a transition (p, γ, q).

Notice that we can have $\sigma = \epsilon$, in which case $p' = q$, and that all new transitions start at initial states.

Before explaining the intuition for the rule, let us illustrate the procedure by means of an example. Let $\mathcal{P}$ be the pushdown system shown at the top of Fig. 5, and let $\mathcal{A}$ be the automaton recognizing the singleton set $C = \{p_0\gamma_0\gamma_0\}$, shown on the left. The automaton $\mathcal{A}_{pre^*}$ is shown on the right. The saturation procedure adds five additional transitions. The table at the bottom of the figure gives for each new transition of the automaton the transition rule $p\gamma \hookrightarrow p'\sigma$ of the PDS and the path $p' \xrightarrow{\sigma} q$ of the current automaton used to apply the saturation rule. The procedure eventually terminates because the number of possible new transitions is finite.

The intuition for the saturation rule is as follows. Imagine that before adding the transition (p, γ, q) as indicated in the rule, the automaton accepts a configuration $p'\sigma\tau$ by means of a run $p' \xrightarrow{\sigma} q \xrightarrow{\tau} q'$ leading to a final state q'. This means that $p'\sigma\tau \in pre^*(C)$. Since $p\gamma \hookrightarrow p'\sigma$, we have $p\gamma\tau \in pre^*(C)$, and so the automaton

$$P = \{ p_0, p_1, p_2 \} \quad \Gamma = \{ \gamma_0, \gamma_1, \gamma_2 \} \quad \Delta = \{ p_0\gamma_0 \hookrightarrow p_1\gamma_1\gamma_0 \ , \ p_2\gamma_2 \hookrightarrow p_0\gamma_1, \ p_1\gamma_1 \hookrightarrow p_2\gamma_2\gamma_0 \ , \ p_0\gamma_1 \hookrightarrow p_0 \ \}$$



$p\gamma \hookrightarrow p'\sigma$	$p' \xrightarrow{\sigma} q$	New transition
$p_0\gamma_1 \hookrightarrow p_0$	$p_0 \xrightarrow{\varepsilon} p_0$	(p_0, γ_1, p_0)
$p_2\gamma_2 \hookrightarrow p_0\gamma_1$	$p_0 \xrightarrow{\gamma_1} p_0$	(p_2, γ_2, p_0)
$p_1\gamma_1 \hookrightarrow p_2\gamma_2\gamma_0$	$p_2 \xrightarrow{\gamma_2\gamma_0} s_1$	(p_1, γ_1, s_1)
$p_0\gamma_0 \hookrightarrow p_1\gamma_1\gamma_0$	$p_1 \xrightarrow{\gamma_1\gamma_0} s_2$	(p_0, γ_0, s_2)
$p_1\gamma_1 \hookrightarrow p_2\gamma_2\gamma_0$	$p_2 \xrightarrow{\gamma_2\gamma_0} s_2$	(p_1, γ_1, s_2)

Fig. 5 The automata $\mathcal{A}$ (left) and $\mathcal{A}_{pre^*}$ (right)

should also accept $p\gamma\tau$. This is precisely what the saturation rule achieves: after adding the transition (p, γ, q) the automaton has the run $p \xrightarrow{\gamma} q \xrightarrow{\tau} q'$, and so it accepts $p\gamma\tau$.

This argument shows that $pre^*(L(\mathcal{A})) \subseteq L(\mathcal{A}_{pre^*})$ holds. Proving the other inclusion requires some more care, and is outside the scope of this chapter. The proof can be found in [12]. This direction relies on the assumption that $\mathcal{A}$ has no transitions leading to an initial state. Notice that without this assumption the algorithm is incorrect.

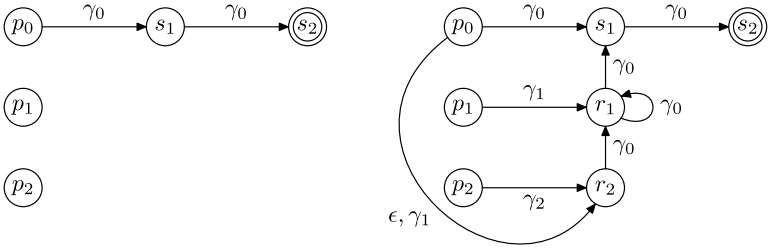
It is clear that the saturation procedure runs in time polynomial in the size of the PDS $\mathcal{P}$ and the automaton $\mathcal{A}$. An efficient implementation and a more careful complexity analysis can be found in [26]:

Theorem 1 ([26]) *Given $\mathcal{P} = (P, \Gamma, \Delta)$ and $\mathcal{A} = (\Gamma, Q, \delta, P, F)$, the automaton $\mathcal{A}_{pre^*}$ can be computed in $O(n_Q^2 n_\Delta)$ time and $O(n_Q n_\Delta + n_\delta)$ space, where $n_Q = |Q|$, $n_\delta = |\delta|$, and $n_\Delta = |\Delta|$.*

17.3.3.2 Computing $post^*(C)$ for a Regular Set C by Saturation

We provide an algorithm for the case in which each transition rule $p\gamma \hookrightarrow p'\sigma$ of Δ satisfies $|\sigma| \leq 2$. This restriction is not essential, but leads to a simpler solution. Moreover, any PDS can be transformed into an equivalent one in this form, and the PDSs derived from RSMs directly satisfy this condition.

$$P = \{ p_0, p_1, p_2 \} \quad \Delta = \{ \; p_0\gamma_0 \hookrightarrow p_1\gamma_1\gamma_0 \;,\; p_2\gamma_2 \hookrightarrow p_0\gamma_1, \;\; p_1\gamma_1 \hookrightarrow p_2\gamma_2\gamma_0 \;,\; p_0\gamma_1 \hookrightarrow p_0 \; \}$$
$$\Gamma = \{ \gamma_0, \gamma_1, \gamma_2 \}$$



$p\gamma \hookrightarrow p'\sigma$	$p \xRightarrow{\gamma} q$	Saturation rule	New transition
$p_2\gamma_2 \hookrightarrow p_0\gamma_1$	$p_2 \xRightarrow{\gamma_2} r_2$	second	(p_0, γ_1, r_2)
$p_0\gamma_0 \hookrightarrow p_1\gamma_1\gamma_0$	$p_0 \xRightarrow{\gamma_0} s_1$	third	(r_1, γ_0, s_1)
$p_0\gamma_1 \hookrightarrow p_0$	$p_0 \xRightarrow{\gamma_1} r_2$	first	(p_0, ε, r_2)
$p_1\gamma_1 \hookrightarrow p_2\gamma_2\gamma_0$	$p_0 \xRightarrow{\gamma_0} r_1$ $(p_0 \xrightarrow{\varepsilon} r_2 \xrightarrow{\gamma_0} r_1)$	third	(r_2, γ_0, r_1)
$p_0\gamma_0 \hookrightarrow p_1\gamma_1\gamma_0$	$p_0 \xRightarrow{\gamma_0} r_1$	third	(r_1, γ_0, r_1)

Fig. 6 The automata $\mathcal{A}$ (left) and $\mathcal{A}_{post^*}$ (right)

Our input is an automaton $\mathcal{A}$ accepting C. Again, we assume that $\mathcal{A}$ has no transitions leading to an initial state. We compute $post^*(C)$ as the language accepted by an automaton $\mathcal{A}_{post^*}$ with ϵ-moves. We denote the relation $(\xrightarrow{\varepsilon})^* \xrightarrow{\gamma} (\xrightarrow{\varepsilon})^*$ by $\xRightarrow{\gamma}$. $\mathcal{A}_{post^*}$ is obtained from $\mathcal{A}$ in two stages:

- Add to $\mathcal{A}$ a new state r for each transition rule $r \in \Delta$ of the form $p\gamma \hookrightarrow p'\gamma'\gamma''$, and a transition (p', γ', r).
- Add new transitions to $\mathcal{A}$ according to the following saturation rules:

> If $p\gamma \hookrightarrow p'\epsilon \in \Delta$ and $p \xRightarrow{\gamma} q$ in the current automaton, then add a transition (p', ϵ, q).
>
> If $p\gamma \hookrightarrow p'\gamma' \in \Delta$ and $p \xRightarrow{\gamma} q$ in the current automaton, then add a transition (p', γ', q).
>
> If $r = p\gamma \hookrightarrow p'\gamma'\gamma'' \in \Delta$ and $p \xRightarrow{\gamma} q$ in the current automaton, then add a transition (r, γ'', q).

Figure 6 shows again the PDS and the automaton from Fig. 5, and, on the right, the automaton $\mathcal{A}_{post^*}$ obtained by applying the algorithm. Since the PDS has two rules of the form $p\gamma \hookrightarrow p'\gamma'\gamma''$, namely $r_1 = p_0\gamma_0 \hookrightarrow p_1\gamma_1\gamma_0$, and $r_2 = p_1\gamma_1 \hookrightarrow p_2\gamma_2\gamma_0$, the first stage of the algorithm adds to $\mathcal{A}_{post^*}$ two new states r_1, r_2, and two new transitions (p_1, γ_1, r_1) and (p_2, γ_2, r_2). In the second stage the algorithm

adds another five transitions. The table at the bottom of the figure gives for each new transition the transition rule $p\gamma \hookrightarrow p'w$ of the PDS, the path $p' \xrightarrow{\gamma} q$ of the current automaton, and the saturation rule used to produce it. Again, an efficient implementation and a more careful complexity analysis can be found in [26].

Theorem 2 ([26]) *Given $\mathcal{P} = (P, \Gamma, \Delta)$ and $\mathcal{A} = (\Gamma, Q, \delta, P, F)$, the automaton $\mathcal{A}_{post^*}$ can be computed in $O(n_P n_\Delta (n_Q + n_\Delta) + n_P n_\delta)$ time and space, where $n_P = |P|$, $n_\Delta = |\Delta|$, $n_Q = |Q|$, and $n_\delta = |\delta|$.*

17.3.4 The Generalized Fair Computation Problem

Section 17.3.2 presents a summarization algorithm for the fair computation problem: given a node p and a set $F \subseteq En$ of repeat nodes, decide whether p has an F-fair computation. We now use saturation to solve a generalized version of the problem: compute the set of *all* configurations of $\mathcal{M}$ having an F-fair computation, i.e., an infinite computation that infinitely often visits nodes in F.

Let $\mathcal{P}_\mathcal{M} = (P_\mathcal{M}, \Gamma_\mathcal{M}, \Delta_\mathcal{M})$ be the PDS associated with $\mathcal{M}$. It is easy to see that $p\sigma$ has an F-fair computation if and only if there exists $p' \in F$ such that $p\sigma \xrightarrow{*} p'\sigma'$ for some $\sigma' \in \Gamma^*_\mathcal{M}$ and $p' \xrightarrow{+} p'\tau$ for some $\tau \in \Gamma^*_\mathcal{M}$. We first compute the set *Rep* of states $q \in F$ such that $q \xrightarrow{+} q\sigma$ for some $\sigma \in \Gamma^*_\mathcal{M}$. The set of configurations that have an infinite fair computation is then equal to $pre^*(Rep\Gamma^*_\mathcal{M})$, which is regular and computable using the construction of Sect. 17.3.3.1.

To compute *Rep* we observe that for every state q we have $q \in Rep$ if and only if $q \in pre^+(q\Gamma^*_\mathcal{M})$, where $pre^+(C) = pre(pre^*(C))$. We construct a finite automaton $\mathcal{A}_{pre^+}$ recognizing $pre^+(q\Gamma^*_\mathcal{M})$ from an automaton $\mathcal{A}$ recognizing C. Since in Sect. 17.3.3.1 we already constructed an automaton $\mathcal{A}_{pre^*}$ recognizing $pre^*(C)$, it suffices to provide another construction doing the same for pre (instead of pre^*). The construction for pre^+ is the result of concatenating the two, i.e., of applying the construction for pre to the result of applying the construction for pre^*.

The construction for pre is, not surprisingly, simpler than the one for pre^*. It starts with some preprocessing. Given an input automaton $\mathcal{A} = (\gamma, Q, \delta, P, F)$, the preprocessing adds to it a fresh set $\hat{P} = \{\hat{p} \mid p \in P_\mathcal{M}\}$ of states, and changes the set of initial states to $\hat{P}$. Formally, the preprocessing returns the automaton $\hat{\mathcal{A}} = (\gamma, Q \cup \hat{P}, \delta, \hat{P}, F)$. After preprocessing, the construction exhaustively applies the following modification of the saturation rule:

> If $p\gamma \hookrightarrow p'\sigma$ and $p' \xrightarrow{\sigma} q$ in the current automaton, then add a transition $(\hat{p}, \gamma, q)$.

(The only change is the substitution of $\hat{p}$ for p in the last line.) With this rule all new transitions start from states in $\hat{P}$, and so new transitions cannot generate further transitions. The correctness of the construction is easy to prove.

This algorithm for computing *Rep*, presented in [12], has polynomial complexity, but can be improved. A more efficient procedure involving Tarjan's algorithm for computing strongly connected components is presented in [26].

Theorem 3 ([26]) *Given $\mathcal{P} = (P, \Gamma, \Delta)$ and a set $F \subseteq P$ of repeat states, the set Rep can be computed in $O(n_P^2 n_\delta)$ time and $O(n_P n_\delta)$ space.*

Recall that the algorithm for the generalized fair computation problem first computes *Rep* and then $pre^*(Rep\Gamma_{\mathcal{M}}^*)$. By Theorem 1, $pre^*(Rep\Gamma_{\mathcal{M}}^*)$ can be computed in $O(|P_{\mathcal{M}}|^2|\Delta_{\mathcal{M}}|)$ time and $O(|P_{\mathcal{M}}||\Delta_{\mathcal{M}}|)$ space, and so the generalized fair computation problem can also be solved within the same time and space bounds.

17.4 Specifying Requirements

In order to specify requirements of programs modeled by RSMs, we first choose a set Σ of *observables*. Each program statement, or transition of the RSM, is labeled with an observation $\sigma \in \Sigma$. A (possibly infinite) execution of the RSM then produces a sequence of observations. In this manner, we can associate a language $L(\mathcal{M})$ with the RSM $\mathcal{M}$ as its observational (linear) semantics. Requirements can be written using linear-time specification formalisms such as Linear Temporal Logic (LTL) (see Chap. 2 [42]). Given an LTL specification φ over the observables Σ, and an RSM model $\mathcal{M}$, the model-checking question is to check whether every sequence in $L(\mathcal{M})$ satisfies the formula φ. To solve this problem, we can compile the negation of the specification into a Büchi automaton $A_{\neg\varphi}$ that accepts all computations that violate φ (see Chap. 4 [38]) and check that the intersection of the languages of $\mathcal{M}$ and $A_{\neg\varphi}$ is empty. This can be solved algorithmically using the analysis algorithms discussed in Sect. 17.3. In this setup, even though the language $L(\mathcal{M})$ is context-free (since the underlying model is a pushdown system), the requirement is given as an ω-regular language.

While many analysis problems such as identifying dead code and accesses to uninitialized variables can be captured as regular requirements, many others require inspection of the stack or matching of calls and returns, and are context-free. These include access control requirements such as "a procedure P should be invoked only if the procedure P' belongs to the call-stack," bounds on stack size such as "if the number of interrupt-handling procedures in the call-stack currently is less than 5, then a property p holds," and correctness specifications using pre- and post-conditions such as "if the property p holds when a procedure P is invoked, the procedure P must return, and the property q holds upon return." When viewed in isolation, each of these requirements is a context-free language, and checking context-free requirements of RSMs (or pushdown systems) is undecidable in general. However, the key feature of these example requirements is that the stacks in the model and the requirement are correlated: while the stacks are not identical, the two synchronize on when to push and when to pop, and are always of the same depth. To

formalize this, we view an execution of the program as a *nested word*, which consists of a linear sequence of states (or observations), augmented with nesting edges connecting calls with matching returns, that impart a tree-like hierarchical structure to the execution. Automata and logics over nested words can be used to express a variety of requirements such as stack-inspection properties, pre- and post-conditions, and interprocedural data-flow properties. Closure properties and decision problems of these automata can then be used for algorithmic verification of procedural programs.

17.4.1 Nested Words

Nested words model data with both linear and hierarchical structure. Here we consider only *infinite* nested words (which can model nonterminating executions of programs).

Given a linear sequence, the hierarchical structure is added using edges that are well nested (that is, they do not cross). We will use edges starting at $-\infty$ and edges ending at $+\infty$ to model "pending" edges. Assume that $-\infty < i < +\infty$ for every integer i. A *matching relation* $\rightsquigarrow$ is a subset of $\{-\infty, 1, 2, \ldots\} \times \{1, 2, \ldots + \infty\}$ such that (1) nesting edges go only forward: if $i \rightsquigarrow j$ then $i < j$; (2) no two nesting edges share a position: for each natural number i, $|\{j \mid i \rightsquigarrow j\}| \leq 1$ and $|\{j \mid j \rightsquigarrow i\}| \leq 1$; and (3) nesting edges do not cross: if $i \rightsquigarrow j$ and $i' \rightsquigarrow j'$ then it is not the case that $i < i' \leq j < j'$.

When $i \rightsquigarrow j$ holds, the position i is called a *call position*. For a call position i, if $i \rightsquigarrow +\infty$, then i is called a *pending call*, otherwise i is called a *matched call*, and the unique position j such that $i \rightsquigarrow j$ is called its *return-successor*. Similarly, when $i \rightsquigarrow j$ holds, the position j is called a *return position*. For a return position j, if $-\infty \rightsquigarrow j$, then j is called a *pending return*, otherwise j is called a *matched return*, and the unique position i such that $i \rightsquigarrow j$ is called its *call-predecessor*. A position i that is neither a call nor a return is called *internal*.

A *nested word* w over an alphabet Σ is a pair $(a_1 a_2 \cdots, \rightsquigarrow)$ such that each a_i is a symbol in Σ, and $\rightsquigarrow$ is a matching relation. Let us denote the set of all nested words over Σ as $NW(\Sigma)$. A *language* of nested words over Σ is a subset of $NW(\Sigma)$.

As an example, consider the program of Fig. 2 again. Suppose we are interested in tracking read/write accesses to the global program variable x. Then, we can choose the following set of symbols for the observables Σ: *rd* to denote a read access to x, *wr* to denote a write access to x, *cl* to denote beginning of a new scope (such as a call to the procedure P_2), *rt* to denote the ending of the current scope, and *sk* to denote all other actions of the program. Note that in any structured programming language, in a given execution, there is a natural nested matching of the symbols *cl* and *rt*. Figure 7 shows a sample execution of the program modeled as a nested word (this execution corresponds to the initial state in which x is 0 and y is 1). For example, the second symbol (labeled *rd*) corresponds to the execution of the test "**if** x", and the next corresponds to the assignment $x := y$. Both these steps

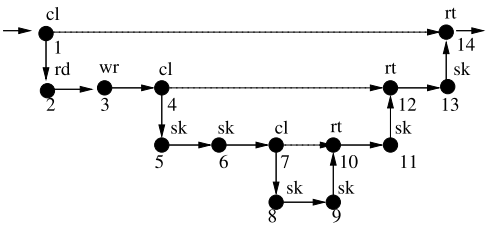


Fig. 7 Sample execution as a nested word

do not involve a change of context, and are internal positions. The procedure P_2 is called at position 4, and this call has a nesting edge to the matching position 12 (labeled *rt*). The subword from position 5 to position 11 encodes the execution of the called procedure. The main benefit of explicitly augmenting the linear structure with the nesting edges is that using nesting edges one can skip calls to a procedure entirely, and continue to trace a local path through the calling procedure. Consider the property that "if a procedure writes to x then it later reads x." This requires keeping track of the context. If we were to model executions as words, the set of executions satisfying this property would be a context-free language of words, and hence, not specifiable in classical temporal logics. Soon we will see that when we model executions as nested words, the set of executions satisfying this property is a regular language of nested words, and is amenable to algorithmic verification.

17.4.2 Nested Word Automata

We define and study finite-state automata as acceptors of nested words. A *nested word automaton* (NWA) is similar to a classical finite-state word automaton, and reads the input from left to right according to the linear sequence. At a call, it can propagate states along both linear and nesting outgoing edges, and at a return, the new state is determined based on states labeling both the linear and nesting incoming edges. Thus, an NWA combines the features of top-down and bottom-up tree automata. It can also be viewed as a restricted form of a pushdown automaton: at a call position, it pushes a symbol onto the stack; at a return position, it pops a symbol from the stack; and at an internal position, it does not update or examine the stack. Thus, the updates to the stack are determined by the call/return structure of the input word, and that's why a nested word automaton is also called a *visibly pushdown* automaton.

In the context of program verification, we are interested in *nondeterministic* NWAs: nondeterminism can arise due to inputs, due to abstraction, or when multiple states/transitions are associated with the same observation. We focus only on automata over infinite words using the Büchi acceptance condition.

A *nondeterministic Büchi nested word automaton* (BNWA) A over an alphabet Σ consists of

- a finite set of states Q,

- a set of initial states $Q_0 \subseteq Q$,
- a set of Büchi states $Q_f \subseteq Q$,
- a finite set of hierarchical states P,
- a set of initial hierarchical states $P_0 \subseteq P$,
- a call transition relation $\delta_c \subseteq Q \times \Sigma \times Q \times P$,
- an internal transition relation $\delta_l \subseteq Q \times \Sigma \times Q$, and
- a return transition relation $\delta_r \subseteq Q \times P \times \Sigma \times Q$.

Given a nested word w, the automaton A starts in an initial state, and reads the nested word from left to right according to the linear order. The state is propagated along the linear edges as in the case of a standard word automaton. However, at a call, the nested word automaton can also propagate a hierarchical state along the outgoing nesting edge. At a return, the new state is determined based on the states propagated along the linear edge as well as along the incoming nesting edge. A pending nesting edge incident upon a pending return is labeled with an initial hierarchical state. The run is accepting if one of the Büchi states repeats infinitely often.

Formally, a run r of the BNWA A over a nested word $w = (a_1 a_2 \cdots, \rightsquigarrow)$ is an infinite sequence $q_i \in Q$, for $i \geq 0$, of states corresponding to linear edges, and a sequence $p_i \in P$, for call positions i, of hierarchical states corresponding to nesting edges, such that $q_0 \in Q_0$, and for each position $i \geq 1$, if i is a call position then $(q_{i-1}, a_i, q_i, p_i) \in \delta_c$; if i is an internal position then $(q_{i-1}, a_i, q_i) \in \delta_l$; if i is a matched return with call-predecessor j then $(q_{i-1}, p_j, a_i, q_i) \in \delta_r$, and if i is a pending return then $(q_{i-1}, p_0, a_i, q_i) \in \delta_r$ for some $p_0 \in P_0$. The run is accepting if $q_i \in Q_f$ for infinitely many indices $i \geq 0$. The automaton A accepts the nested word w if A has some accepting run over w. The language $L(A)$ is the set of nested words A accepts. A set L of nested words is *ω-regular* iff there is a BNWA A such that $L(A) = L$.

17.4.2.1 RSMs as NWAs

An RSM can be interpreted as a nested word automaton. Consider an RSM $\mathcal{M} = (A_1, \ldots, A_k)$ with components $A_i = (N_i, B_i, Y_i, En_i, Ex_i, \delta_i)$. For the corresponding NWA $A_{\mathcal{M}}$, for each component A_i, for every node, call-port, and return port of A_i, there is a corresponding linear state in $A_{\mathcal{M}}$. The set of hierarchical states is the set of boxes of all the components. The entry nodes of the main component are the initial states, and the NWA does not rely on initial hierarchical states (since there will be no pending returns in the nested words it generates). For every transition $u \to v$ of each component A_i, there is a corresponding internal transition in $A_{\mathcal{M}}$. For every call port (en, b) of A_i, the NWA has a call transition from the state (en, b) to the state en (corresponding to the entry node of the component $A_{Y(b)}$) propagating the hierarchical state b along the nesting edge. For every return port (ex, b) of A_i, the NWA has a return transition to the state (ex, b) from the state ex (corresponding to the exit node of the component $A_{Y(b)}$) provided the hierarchical state along the incoming nesting edge is b. The labels on the transitions correspond

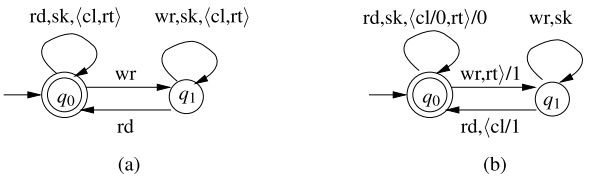


Fig. 8 Using NWA to specify program requirements

to observations suitable for the analysis problem. In the example corresponding to Fig. 7, each call transition is labeled with the symbol cl, each return transition is labeled with the symbol rt, and each internal transition is labeled with either rd, wr, or sk, depending on the type of statement executed. The NWA is augmented with a Büchi acceptance condition if needed (for instance, to ensure fair resolution of choice when nondeterminism is used for abstraction).

17.4.2.2 NWAs for Requirements

The requirements of a program can also be described as an ω-regular language of nested words. Let us revisit the example used in Fig. 7. Suppose we want to specify that each write to x is followed by some read of x. We will consider two variations of this requirement.

First, suppose we want to specify that a symbol wr is followed by rd, without any reference to the procedural context. This can be captured by standard word automata, and also by NWAs. Figure 8(a) shows the two-state (deterministic) NWA for the requirement. We use the prefix $\langle$ with a symbol to indicate a call transition, and the suffix $\rangle$ with a symbol to indicate a return transition. Call and return transitions also have associated hierarchical states. In this example, hierarchical states are not needed.

Now suppose, we want to specify that if a procedure writes to x, then the same invocation should read it before it returns. That is, between every pair of matching call and return, along the local path obtained by deleting every enclosed well-matched subword between a call and its matching return, every wr is followed by rd. Viewed as a property of words, this is not a regular language, and thus, not expressible in the classical specification languages. However, over nested words, this can easily be specified using an NWA, see Fig. 8(b). The initial state is q_0, which has no pending obligations, and is the only final state. The hierarchical states are $\{0, 1\}$, where 0 is the initial state. The state q_1 means that along the local path of the current scope, a write-access has been encountered with no following read access. While processing the call, the automaton remembers the current state by propagating 0 or 1 along the nesting edge, and starts checking the requirement for the called procedure by transitioning to the initial state q_0. While processing internal read/write symbols, it updates the state as in the finite-state word automaton of case (a). At a return, if the current state is q_0 (meaning the current context satisfies the desired requirement), it restores the state of the calling context. Note that there are no return transitions from the state q_1, and this means that if a return position is encountered while in state q_1, the automaton rejects the input word.

We now review some key properties of nested word automata that are useful in their application to model checking.

17.4.2.3 Closure Properties

The class of ω-regular (and regular) languages of nested words is closed under a variety of operations including union, intersection, complementation, prefixes, suffixes, concatenation, Kleene-*, and language homomorphisms. For verification, the most relevant operation is *language intersection*: given two BNWAs A_1 and A_2, one can construct a product BNWA A such that $L(A) = L(A_1) \cap L(A_2)$. If A_1 captures the set of nested words generated by an RSM, and A_2 captures the set of nested words that violate a desired correctness requirement, then verification corresponds to checking non-emptiness of the language of A. The product construction for NWAs is a simple extension of the product construction for finite (word) automata. A linear state of A is a pair of linear states of A_1 and A_2, and a hierarchical state of A is a pair of hierarchical states of A_1 and A_2. The call/internal/return transitions synchronize the transitions of A_1 and A_2 on a common input symbol, and update the two state components. Ensuring that Büchi acceptance conditions of both are satisfied can be done the same way as in the product construction for Büchi automata (see Chap. 4 [38]). It is worth noting that nested word automata can also be complemented and determinized. Determinization requires maintaining a set of "summaries" that capture executions of the nondeterministic automaton on the subword between a call and its matching return, and the acceptance condition needed is a parity condition over states that repeat infinitely often at the "top level" of the input word (see [6] for details). The complexity of determinization as well as of complementation is exponential.

17.4.2.4 Decision Problems

The *emptiness* problem for NWAs (given a BNWA A, is $L(A) = \emptyset$?) is solvable in polynomial time (in time cubic in the size of the automaton). The technique is the same as the one used in solving the fair computation problem for pushdown systems discussed in Sect. 17.3.2. Problems such as *universality* (given a BNWA A, is $L(A) = \Sigma^\omega$?), language *inclusion* (given BNWAs A_1 and A_2, is $L(A_1) \subseteq L(A_2)$?), and language *equivalence* (given BNWAs A_1 and A_2, is $L(A_1) = L(A_2)$?) can all be solved in EXPTIME by employing the complementation construction. Note that these problems are undecidable for pushdown automata (or context-free languages). Thus, given two RSMs, checking whether they generate the same sets of words is undecidable, while checking whether they generate the same sets of nested words is decidable. The latter is a stronger requirement which considers two executions equivalent when the two produce the same sequences of observations, and also agree on entries to and exits from procedural contexts.

17.4.2.5 MSO Equivalence

For word languages, the notion of regularity has many equivalent characterizations using finite automata, monadic second-order logic, and regular expressions. The notion of regularity for nested words also turns out to be robust. In particular, the monadic second order logic (MSO) of nested words has the same expressiveness as nested word automata. The vocabulary of nested sequences includes the linear successor and the matching relation $\rightsquigarrow$. In order to model pending edges, we will use two unary predicates `call` and `ret` corresponding to call and return positions. The *monadic second-order logic of nested words* is given by the syntax:

$$\begin{aligned}\phi &:= a(x) \mid X(x) \mid \texttt{call}(x) \mid \texttt{ret}(x) \mid x \\ &= y+1 \mid x \rightsquigarrow y \mid \phi \vee \phi \mid \neg\phi \mid \exists x.\phi \mid \exists X.\phi,\end{aligned}$$

where $a \in \Sigma$, x, y are first-order variables, and X is a second-order variable. The semantics is defined over nested words in a natural way. The first-order variables are interpreted over positions of the nested word, while set variables are interpreted over sets of positions. The formula $a(x)$ holds if the symbol at the position interpreted for x is a, $\texttt{call}(x)$ holds if the position interpreted for x is a call, $x = y + 1$ holds if the position interpreted for y is (linear) next to the position interpreted for x, and $x \rightsquigarrow y$ holds if the positions x and y are related by a nesting edge. For example,

$$\forall x.\big(\ \texttt{call}(x)\ \rightarrow \exists y.\ x \rightsquigarrow y\ \big)$$

holds in a nested word iff it has no pending calls;

$$\forall x.\forall y.\ \big(a(x) \wedge x \rightsquigarrow y\big) \Rightarrow b(y)$$

holds in a nested word iff for every matched call labeled a, the corresponding return-successor is labeled b.

For a sentence ϕ (a formula with no free variables), the language ϕ defines is the set of all nested words that satisfy ϕ. It turns out that: a language L of nested words over Σ is ω-regular iff there is an MSO sentence ϕ over Σ that defines L.

17.4.3 Temporal Logics

Over infinite words, Linear Temporal Logic (LTL) has long been considered the temporal logic of choice for program verification, not only because its temporal operators offer the right abstraction for reasoning about events over time, but also because it provides a good balance between expressiveness (first-order complete), conciseness (can be exponentially more succinct compared to automata), and the complexity of model checking (time linear in the size of the finite transition system, and PSPACE in the size of the temporal formula). This has motivated the study of temporal logics over nested words such as CaRet [4] and NWTL [1]. We briefly review these logics in this section.

Let us first recall the syntax and semantics of LTL (see Chap. 2 [42]). Given a set AP of atomic propositions, a formula of propositional LTL is built from atomic propositions, logical connectives (such as conjunction $\wedge$, disjunction $\vee$, negation $\neg$, implication $\rightarrow$), and temporal operators (such as next $\bigcirc$, always $\Box$, eventually $\Diamond$, and until $\mathcal{U}$). An LTL formula is evaluated with respect to an infinite sequence $w = a_1 a_2 \cdots$ over $\Sigma = 2^{AP}$, that is, each observation a_j is an assignment of truth values to the propositions in AP. The semantics of LTL is defined using the satisfaction relation $(w, j) \models \phi$, which means that the formula ϕ is satisfied at position j in the model w. Example rules for evaluation are: $(w, j) \models p$, for an atomic proposition p, if the observation w_j assigns the value 1 to p; $(w, j) \models \bigcirc\phi$ if $(w, j+1) \models \phi$; $(w, j) \models \Box\phi$ if $(w, k) \models \phi$ for every position $k \geq j$; and $(w, j) \models \phi_1 \mathcal{U} \phi_2$ if there exists a position $k \geq j$ such that $(w, k) \models \phi_2$ and $(w, l) \models \phi_1$ for all positions $j \leq l < k$.

In the revised setting of nested words, a formula is interpreted over a nested word w over the set $\Sigma = 2^{AP}$ of observations. To motivate the definition of new temporal operators, let us examine the nested word shown in Fig. 7. Notice that unlike a linear sequence, the graph-like structure of a nested word means that one can define different kinds of paths. If we ignore the nesting edges, and focus on the linear sequence of positions, we obtain the *linear* path, and we can continue to interpret LTL operators over this linear path. In this example, the sequence $1, 2, 3, 4, \ldots, 13, 14$ of positions forms the linear path. Suppose we want to express the requirement that, along a global program execution, every write to a variable is followed by a read (see the automaton in Fig. 8(a)). If wr and rd denote the atomic propositions that capture write and read operations, respectively, then the requirement is expressed by the LTL formula:

$$\Box [\ wr \ \rightarrow \Diamond rd\ ].$$

17.4.3.1 Abstract Next

In a nested word, a call position has two successors: a linear edge to the next position, and a nesting edge to the matching return. This motivates adding, besides the original LTL operator $\bigcirc$ corresponding to the linear successor, another next operator, called *abstract-next*, denoted $\bigcirc^a$. Its semantics is defined by the rule:

$(w, j) \models \bigcirc^a \phi$ holds if the position j is a call position, has a matching return position l (that is, $j \rightsquigarrow l$), and $(w, l) \models \phi$.

It is easy to establish that the abstract-next operator is not definable in LTL. In the classical verification formalisms such as Hoare logic, correctness of procedures is expressed using pre- and post-conditions. Partial correctness of a procedure P specifies that if the pre-condition p holds when the procedure P is invoked, if the procedure terminates, the post-condition q is satisfied upon return. Total correctness, in addition, requires the procedure to terminate. Assume that all calls to the procedure P are characterized by the proposition cl_P. Then, the requirement

$$\Box \left[(cl_P \ \wedge \ p) \ \rightarrow \bigcirc^a q \ \right]$$

expresses total correctness, while

$$\Box\left[\left(cl_P \wedge p \wedge \bigcirc^a \texttt{True}\right) \rightarrow \bigcirc^a q\right]$$

expresses partial correctness.

17.4.3.2 Abstract Paths

An *abstract* path in a nested word w is a sequence of positions $i_1, i_2, \ldots i_j$ such that, for each $1 \leq l < j$, either i_l is a call position with matching return position i_{l+1}, or i_l is an internal or a return position and i_{l+1} equals $i_l + 1$ and is not a return position. For a nested word that models an execution of a procedural program, the abstract path starting at a position inside a procedure P is obtained by successive applications of internal and nesting edges, and skips over invocations of other procedures called from P. In the nested word of Fig. 7, examples of abstract paths are 1, 14, and 2, 3, 4, 12, 13, and 5, 6, 7, 10, 11, and 8, 9. We can now define the abstract versions of temporal operators such as *abstract-always* $\Box^a$, *abstract-eventually* $\Diamond^a$, and *abstract-until* $\mathcal{U}^a$. The semantics of these operators is defined by interpreting them over abstract paths. For example,

$(w, j) \models \phi_1 \mathcal{U}^a \phi_2$ if there exists an abstract path $j = i_1, i_2, \ldots i_k$ such that $(w, i_k) \models \phi_2$ and $(w, i_l) \models \phi_1$ for all $1 \leq l < k$.

That is, $\phi_1 \mathcal{U}^a \phi_2$ holds if there is abstract path leading to a position satisfying ϕ_2 such that at all preceding positions along this abstract path ϕ_1 holds. We can use these abstract modalities to specify context-bounded requirements. Let us revisit the requirement that if a procedure writes to a variable, then it (that is, the same invocation of the same procedure) will later read it (see the NWA of Fig. 8(b)). The requirement is expressed by the following formula over abstract paths:

$$\Box\left[\, wr \rightarrow \Diamond^a rd \,\right].$$

17.4.3.3 Summary Paths

A *summary path* between positions i and j, with $i < j$, of a nested word w is a sequence $i = i_1, i_2 \ldots i_k = j$ of positions such that for $1 \leq l < k$, if i_l is a matched call with a matching return position $r \leq j$ then $i_{l+1} = r$, else $i_{l+1} = i_l + 1$. Intuitively, a summary path between i and j is the "shortest" path from i to j that one can construct using linear and nesting edges. For example, in the nested word of Fig. 7, the summary path between positions 2 and 14 is the sequence 2, 3, 4, 12, 13, 14, while the summary path between positions 2 and 11 is the sequence 2, 3, 4, 5, 6, 7, 10, 11. The summary-versions of temporal operators, such as summary-until $\mathcal{U}^\sigma$, are defined by interpreting the temporal modalities over the summary paths. While not particularly natural for specifying program requirements, interest in the summary paths stems from their theoretical expressiveness: the expressiveness of the logic with abstract-next, and its past dual, abstract-previous, and summary-until, and its

past dual, summary-since, coincides exactly with first-order logic over nested words (that is, logic with first-order variables, quantification over first-order variables, logical connectives, binary predicates $x = y + 1$, $x < y$, $x \rightsquigarrow y$, and unary predicates corresponding to `call`, `ret`, and atomic propositions) [1]. This result is the analog of the result that the expressiveness of LTL coincides with first-order logic over words. Global, abstract, and other versions of temporal modalities are definable using first-order logic over nested words, and this implies that requirements about abstract paths can be defined using modalities over summary paths. It seems unlikely that a similar completeness result holds for abstract modalities (more specifically, it is conjectured, but not proved, that the logic CaRet [4] is not first-order complete).

17.4.3.4 Model Checking

Chapter 4 [38] discusses the tableau-based approach to checking satisfiability and model checking of LTL. This approach can be extended to temporal logics over nested words. In the sequel, we use NWTL to denote the logic with all the connectives we have discussed so far, and also their past duals. Given an NWTL formula φ, we can construct a BNWA A_φ such that (1) $L(A_\varphi)$ contains exactly those nested words that satisfy φ, and (2) the size of A_φ is $2^{O(|\varphi|)}$. To check whether φ is satisfiable, we can test whether the language of A_φ is nonempty, and to check whether all executions of an RSM $\mathcal{M}$ satisfy the NWTL specification φ, we can test language-emptiness of the product of the automata $A_\mathcal{M}$ and $A_{\neg\varphi}$. Both satisfiability and model-checking problems for NWTL are EXPTIME-complete.

The construction of the BNWA A_φ corresponding to the NWTL formula φ follows the same recipe as the tableau construction for LTL discussed in Chap. 4 [38]. We first define the set $Closure(\varphi)$ of formulas; the linear and hierarchical states of A_φ are subsets of $Closure(\varphi)$ that satisfy local consistency requirements; the transitions of A_φ are defined so that next-time requirements are correctly propagated along the linear edges, and abstract-next-time requirements are correctly propagated along the nesting edges; and each until-formula in the closure gives a Büchi acceptance condition that ensures eventual fulfillment of the until obligations (this results in a *generalized* Büchi acceptance condition, which can be translated into a Büchi acceptance condition by introducing a counter as described in Chap. 4 [38]). We refer the reader to [1] for details, but illustrate the essence of the construction by focusing on *abstract-until* formulas of the form $\phi_1 \mathcal{U}^a \phi_2$.

The closure contains propositions `call`, `ret`, and `int`, that indicate the position types. Additionally, a proposition `top` is used to indicate whether the current position is "top level": a position i of a nested word w is top level if it is not within a pair of matching call-return positions, that is, there are no positions j and k such that $j < i < k$ and $j \rightsquigarrow k$.

The closure rule for the abstract-until formula says that if $\phi_1 \mathcal{U}^a \phi_2$ is in $Closure(\varphi)$ then so are the formulas ϕ_1, ϕ_2, $\bigcirc(\phi_1 \mathcal{U}^a \phi_2)$ and $\bigcirc^a(\phi_1 \mathcal{U}^a \phi_2)$. The size of the closure is linear in $|\varphi|$.

States correspond to subsets of the closure that satisfy consistency requirements. Sample consistency requirements on a state $\Phi \subseteq Closure(\varphi)$ are: exactly one of `call`, `ret`, and `int` belongs to Φ, and $\phi_1 \mathcal{U}^a \phi_2 \in \Phi$ iff either $\phi_2 \in \Phi$, or ($\phi_1 \in \Phi$ and `call` $\in \Phi$ and $\bigcirc^a(\phi_1 \mathcal{U}^a \phi_2) \in \Phi$) or ($\phi_1 \in \Phi$ and `call` $\notin \Phi$ and $\bigcirc$`ret` $\notin \Phi$ and $\bigcirc(\phi_1 \mathcal{U}^a \phi_2) \in \Phi$). Note this rule for the abstract-until formula captures its semantics inductively: to satisfy the formula $\phi_1 \mathcal{U}^a \phi_2$ at a position either ϕ_2 is satisfied in that position, or at a call position, ϕ_1 is satisfied and the formula is propagated along the nesting edge, or at a return/internal position, ϕ_1 is satisfied and the formula is propagated along the linear edge, provided the linear successor is not a return.

The transitions of the automaton ensure that the desired propagation expressed by next and abstract-next formulas in a state is enforced. If there is an internal transition from state Φ to state Ψ, then it must be the case that `top` $\in \Phi$ iff `top` $\in \Psi$ and for each $\bigcirc\psi \in Closure(\varphi)$, $\psi \in \Psi$ iff $\bigcirc\psi \in \Phi$. If there is a call transition from state Φ to state Φ_l while propagating state Φ_h on the nesting edge, then it must be the case that either none of Φ, Ψ_l and Ψ_h contain `top`, or `top` $\in \Phi$ and exactly one of Ψ_l and Ψ_h contains `top`; and for each $\bigcirc\psi \in Closure(\varphi)$, $\psi \in \Psi_l$ iff $\bigcirc\psi \in \Phi$; and for each $\bigcirc^a\psi \in Closure(\varphi)$, $\psi \in \Psi_h$ iff $\bigcirc^a\psi \in \Phi$. Finally, if there is a return transition to state Ψ from state Φ_l using the incoming hierarchical state Φ_h, then it must be the case that `top` $\notin \Phi_l$, and `top` $\in \Phi_h$ iff `top` $\in \Psi$; for each $\bigcirc\psi \in Closure(\varphi)$, $\psi \in \Psi$ iff $\bigcirc\psi \in \Phi_l$; and for each $\bigcirc^a\psi \in Closure(\varphi)$, $\psi \in \Phi_h$ iff $\psi \in \Phi_l$.

The Büchi acceptance condition to ensure the eventual fulfillment of the abstract-until formula $\phi_1 \mathcal{U}^a \phi_2$ demands that some state Φ exists such that `top` $\in \Phi$ and either $\phi_2 \in \Phi$ or $\phi_1 \mathcal{U}^a \phi_2 \notin \Phi$ repeats infinitely often. This is based on the fact that the fulfillment of an abstract-until can be delayed forever by the propagation rules only along an abstract path that contains only top-level positions.

17.5 Bibliographical Remarks

17.5.1 Summarization

Two early papers proposing general frameworks for computing procedure summaries in the context of inter-procedural program analysis are [25] by Cousot and Cousot and [49] by Sharir and Pnueli. There is a lot of subsequent work aimed at investigating efficient techniques for various kinds of abstract domains to account for data manipulated by the program, and designing efficient and precise algorithmic techniques for special classes of properties ([30, 40, 44, 46]). In particular, Reps et al. propose in [44] efficient algorithms for inter-procedural data-flow analysis based on graph reachability that is similar to checking reachability in pushdown systems. The tool Bebop by Ball and Rajamani [9] allows verification of sequential Boolean programs with procedure calls using basically the reachability analysis algorithm of [44]. The model of recursive state machines was defined in [2] as a generalization of the model of hierarchical state machines [7], and this work gives a detailed

analysis of the complexity of solving reachability, fair computation, and model-checking problems for temporal logics such as LTL and CTL*, based on summarization. Working directly with RSMs allows an understanding of the dependence of the computational complexity on the number of entry/exit nodes per component.

17.5.2 Saturation

The regularity of $pre^*(L)$ for a regular language L seems to have been first observed by Büchi in his work on regular canonical systems (see Chap. 5 of [15]), and has been rediscovered many times in slightly different contexts, for instance by Caucal in [21] and by Book and Otto in [10]. Book and Otto also present the saturation algorithms for monadic string-rewriting systems, a model closely related to PDSs.

Saturation algorithms for computing sets of forward- and backward-reachable configurations of PDSs were presented by Bouajjani et al. and Finkel et al. [12, 29]. Efficient versions with a detailed complexity analysis were obtained by Esparza et al. [26] (see also [47]). Symbolic versions of the algorithms were implemented in the MOPED tool by Schwoon and applied to verification problems of Linux drivers [28, 47]. The jMOPED tool adds to MOPED a front-end that transforms Java programs into extended pushdown systems and allows MOPED [52] to be applied.

The saturation technique has been extended in a number of ways. We briefly summarize some of the contributions.

Bouajjani et al. extend the technique to *alternating pushdown systems*, and apply the algorithms to the global[1] model-checking problem of CTL [12]. They show for a given CTL formula ϕ and a PDS $\mathcal{P}$ how to compute the set of all configurations of $\mathcal{P}$ satisfying ϕ. A different extension leading to a similar algorithm for CTL* is described by Esparza et al. in [27]. An efficient algorithm for CTL model checking based on solving emptiness of alternating Büchi pushdown automata has been defined in [50].

Reps et al. show how to apply saturation to *weighted pushdown systems*, in which transition rules are labeled with elements of an idempotent semiring [45]. The saturation algorithm is extended so that it returns not only the sets $pre^*(C)$ and $post^*(C)$, but for each configuration c in them the total weight of the paths leading from c to C or from C to c, respectively. The extensions are implemented in the Weighted Automata Library WALi [37]. While the original motivation of this work was to obtain a general framework for inter-procedural data-flow analysis, the developed framework and algorithms were shown to be also useful for other applications, like modeling and verifying trust-management systems [36].

Cachat describes a saturation algorithm for computing the *attractor* of a regular set C of configurations of a *pushdown game system* [20]. A pushdown game system is a PDS whose states are partitioned into two sets under the control of two different

[1] Here *global model checking* means computing the set of all states in a given model that satisfy some given formula.

players. A play is a sequence of configurations, where the successor of the current configuration is decided by the player owning its control state. The attractor is the set of configurations such that the first player can force the play to visit C. Hague and Ong extend Cachat's ideas to algorithms for computing the winning regions of a given parity game [32], and for a given PDS $\mathcal{P}$ and a given formula ϕ of the μ-calculus the set of all configurations of $\mathcal{P}$ satisfying ϕ [33].

Higher-order pushdown systems (HPDSs) generalize PDSs by allowing nested stacks, i.e., stacks whose elements can be stacks themselves. Bouajjani and Meyer extend the saturation algorithm to HPDSs with one control state, also called higher-order context-free processes [14]. Hague and Ong extend the results to general HPDSs [31]. Seth gives an alternative construction for order 2 [48].

17.5.3 Temporal Logic Model Checking

Model checking of pushdown systems has been studied extensively for both linear- and branching-time requirements (see e.g. [2, 12, 19, 26, 27, 29, 43, 55]). The decidability of the model-checking problem of pushdown systems for the propositional μ-calculus (which subsumes in expressiveness regular propositional temporal logics such as LTL and CTL*) follows from results in [39]. However, the model-checking algorithm derived from this result, which is based on a reduction to the satisfiability problem of the monadic second-order logic of two successors, has a non-elementary complexity. In [16], an elementary algorithm is provided for the class of context-free processes (equivalent to pushdown systems with a single control state) and the alternation-free (branching-time) propositional μ-calculus. Basically, this algorithm generalizes the summarization construction as it is based on computing pairs of pre- and post-conditions of a process. The algorithm has been extended to the full class of pushdown systems (but still for alternation-free μ-calculus) in [17], and then later to the full propositional μ-calculus, but only for context-free processes, in [18]. The algorithms defined in this work have been implemented in a tool called "The Fixpoint-Analysis Machine" [51] that has been used in practice for tackling various problems such as intra/inter-procedural data-flow analysis, model checking, and behavioral equivalence checking. The first elementary model-checking algorithm for the full class of pushdown systems and the full propositional μ-calculus has been defined in [53]. The algorithm is based on solving pushdown parity games. A global model-checking algorithm for this general case has been provided first in [43]. In [53], the model-checking problem of pushdown systems for the full μ-calculus is shown to be EXPTIME-complete. In [54], it is shown that the problem is EXPTIME-complete even for CTL, and that it is PSPACE-complete for the EF fragment. In [12], the problem is shown to be EXPTIME-complete for LTL and the linear-time propositional μ-calculus.

Even though the general problem of checking context-free properties of pushdown automata is undecidable, algorithmic solutions have been proposed for checking many different kinds of non-regular properties. For example, numerical properties have been considered in [11, 13] where model-checking algorithms are defined

for extension of temporal logics with constraints on the number of occurrences of events/states along computations. These logics allow for instance properties such as "*between every pair of events a and b, there is the same number of c's as there are d's to be expressed*". The model-checking algorithms proposed for these logics are based on reductions to the satisfiability of Presburger arithmetics, using the fact that Parikh-images of context-free languages are semi-linear sets [41].

Non-numerical properties have also been considered in several works. For instance, access control requirements such as "*a module A should be invoked only if the module B belongs to the call-stack*", and bounds on stack size such as "*if the number of interrupt-handlers in the call-stack currently is less than 5, then a property p holds*" require inspection of the stack, and decision procedures for certain classes of stack properties have been proposed [23, 27, 35].

The idea of explicit modalities that can refer to the matching structure of calls and returns first appears in the temporal logic CaRet [4]. Subsequently, the model of visibly pushdown automata [5] and the theory of regular languages of nested words [6] were proposed as a unifying basis to explain which class of properties are algorithmically checkable against pushdown models. [1] defines the temporal logic NWTL, and presents a systematic study of linear temporal logics over nested words. [24] describes a specification language called Pal that extends the query language of the software model checker Blast [34] for writing nested word monitors, along with a tool to annotate C code.

The nested structure on words can be extended to trees, and automata on nested trees are studied in [3]. A version of the μ-calculus on nested structures has been defined in [3], and is shown to be more powerful than the standard μ-calculus, while at the same time remaining robust and tractable.

References

1. Alur, R., Arenas, M., Barcelo, P., Etessami, K., Immerman, N., Libkin, L.: First-order and temporal logics for nested words. In: Proc. of the Symp. on Logic in Computer Science (LICS), pp. 151–160. IEEE, Piscataway (2007)
2. Alur, R., Benedikt, M., Etessami, K., Godefroid, P., Reps, T.W., Yannakakis, M.: Analysis of recursive state machines. Trans. Program. Lang. Syst. **27**(4), 786–818 (2005)
3. Alur, R., Chaudhuri, S., Madhusudan, P.: A fixpoint calculus for local and global program flows. In: Proc. of the Symp. on Principles of Programming Languages (POPL), pp. 153–165. ACM, New York (2006)
4. Alur, R., Etessami, K., Madhusudan, P.: A temporal logic of nested calls and returns. In: Jensen, K., Podelski, A. (eds.) Proc. of the Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 2988, pp. 467–481. Springer, Heidelberg (2004)
5. Alur, R., Madhusudan, P.: Visibly pushdown languages. In: Proc. of the Symp. on Theory of Computing (STOC), pp. 202–211. ACM, New York (2004)
6. Alur, R., Madhusudan, P.: Adding nesting structure to words. J. ACM **56**(3), 16:1–16:43 (2009)
7. Alur, R., Yannakakis, M.: Model checking of hierarchical state machines. Trans. Program. Lang. Syst. **23**(3), 1–31 (2001)

8. Ball, T., Levin, V., Rajamani, S.K.: A decade of software model checking with SLAM. Commun. ACM **54**(7), 68–76 (2011)
9. Ball, T., Rajamani, S.K.: Bebop: a symbolic model checker for boolean programs. In: Havelund, K., Penix, J., Visser, W. (eds.) Proc. of the Intl. Symp. on Model Checking of Software (SPIN). LNCS, vol. 1885, pp. 113–130. Springer, Heidelberg (2000)
10. Book, R., Otto, F.: String-Rewriting Systems. Springer, Heidelberg (1993)
11. Bouajjani, A., Echahed, R., Habermehl, P.: On the verification problem of nonregular properties for nonregular processes. In: Proc. of the Symp. on Logic in Computer Science (LICS), pp. 123–133. IEEE, Piscataway (1995)
12. Bouajjani, A., Esparza, J., Maler, O.: Reachability analysis of pushdown automata: application to model checking. In: Mazurkiewicz, A., Winkowski, J. (eds.) Proc. of the Intl. Conf. on Concurrency Theory (CONCUR). LNCS, vol. 1243, pp. 135–150. Springer, Heidelberg (1997)
13. Bouajjani, A., Habermehl, P.: Constrained properties, semilinear systems, and Petri nets. In: Montanari, U., Sassone, V. (eds.) Proc. of the Intl. Conf. on Concurrency Theory (CONCUR). LNCS, vol. 1119, pp. 481–497. Springer, Heidelberg (1996)
14. Bouajjani, A., Meyer, A.: Symbolic reachability analysis of higher-order context-free processes. In: Lodaya, K., Mahajan, M. (eds.) Proc. of the Intl. Conf. on Foundations of Software Technology and Theoretical Computer Science (FSTTCS). LNCS, vol. 3328, pp. 135–147 (2004)
15. Büchi, J.R.: In: Siefkes, D. (ed.) Finite Automata, Their Algebras and Grammars. Springer, Heidelberg (1988)
16. Burkart, O., Steffen, B.: Model checking for context-free processes. In: Cleaveland, W. (ed.) Proc. of the Intl. Conf. on Concurrency Theory (CONCUR). LNCS, vol. 630, pp. 123–137. Springer, Heidelberg (1992)
17. Burkart, O., Steffen, B.: Pushdown processes: parallel composition and model checking. In: Jonsson, B., Parrow, J. (eds.) Proc. of the Intl. Conf. on Concurrency Theory (CONCUR). LNCS, vol. 836, pp. 98–113. Springer, Heidelberg (1994)
18. Burkart, O., Steffen, B.: Model checking the full modal mu-calculus for infinite sequential processes. In: Degano, P., Gorrieri, R., Marchetti-Spaccamela, A. (eds.) Proc. of the Intl. Colloquium on Automata, Languages and Programming (ICALP). LNCS, vol. 1256, pp. 419–429. Springer, Heidelberg (1997)
19. Burkart, O., Steffen, B.: Model checking the full modal mu-calculus for infinite sequential processes. Theor. Comput. Sci. **221**(1–2), 251–270 (1999)
20. Cachat, T.: Symbolic strategy synthesis for games on pushdown graphs. In: Widmayer, P., Ruiz, F.T., Bueno, R.M., Hennessy, M., Eidenbenz, S., Conejo, R. (eds.) Proc. of the Intl. Colloquium on Automata, Languages and Programming (ICALP). LNCS, vol. 2380, pp. 704–715. Springer, Heidelberg (2002)
21. Caucal, D.: On the regular structure of prefix rewriting. Theor. Comput. Sci. **106**(1), 61–86 (1992)
22. Chaki, S., Gurfinkel, A.: BDD-based symbolic model checking. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
23. Chatterjee, K., Ma, D., Majumdar, R., Zhao, T., Henzinger, T., Palsberg, J.: Stack size analysis for interrupt driven programs. Inf. Comput. **194**(2), 144–174 (2004)
24. Chaudhuri, S., Alur, R.: Instrumenting C programs with nested word monitors. In: Bosnacki, D., Edelkamp, S. (eds.) Proc. of the Intl. Symp. on Model Checking of Software (SPIN). LNCS, vol. 4595. Springer, Heidelberg (2007)
25. Cousot, P., Cousot, R.: Static determination of dynamic properties of recursive procedures. In: IFIP WG2.2 Conference on Formal Description of Programming Concepts, pp. 237–277. North-Holland, Amsterdam (1978)
26. Esparza, J., Hansel, D., Rossmanith, P., Schwoon, S.: Efficient algorithms for model checking pushdown systems. In: Emerson, E.A., Sistla, A.P. (eds.) Proc. of Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 1855, pp. 232–247. Springer, Heidelberg (2000)

27. Esparza, J., Kucera, A., Schwoon, S.: Model checking LTL with regular valuations for pushdown systems. Inf. Comput. **186**(2), 355–376 (2003)
28. Esparza, J., Schwoon, S.: A BDD-based model checker for recursive programs. In: Berry, G., Comon, H., Finkel, A. (eds.) Proc. of Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 2102, pp. 324–336. Springer, Heidelberg (2001)
29. Finkel, A., Willems, B., Wolper, P.: A direct symbolic approach to model checking pushdown systems. Electron. Notes Theor. Comput. Sci. **9**, 27–37 (1997)
30. Gulwani, S., Tiwari, A.: Computing procedure summaries for interprocedural analysis. In: Nicola, R.D. (ed.) Proc. of the European Symp. on Programming (ESOP). LNCS, vol. 4421, pp. 253–267. Springer, Heidelberg (2007)
31. Hague, M., Ong, C.H.L.: Symbolic backwards-reachability analysis for higher-order pushdown systems. In: Seidl, H. (ed.) Proc. of the Intl. Conf. on Foundations of Software Science and Computational Structures (FoSSaCS). LNCS, vol. 4423, pp. 213–227. Springer, Heidelberg (2007)
32. Hague, M., Ong, C.H.L.: Winning regions of pushdown parity games: a saturation method. In: Bravetti, M., Zavattaro, G. (eds.) Proc. of the Intl. Conf. on Concurrency Theory (CONCUR). LNCS, vol. 5710, pp. 384–398. Springer, Heidelberg (2009)
33. Hague, M., Ong, C.H.L.: A saturation method for the modal μ-calculus over pushdown systems. Inf. Comput. **209**(5), 799–821 (2011)
34. Henzinger, T., Jhala, R., Majumdar, R., Necula, G., Sutre, G., Weimer, W.: Temporal-safety proofs for systems code. In: Brinksma, E., Larsen, K.G. (eds.) Proc. of the Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 2404, pp. 526–538. Springer, Heidelberg (2002)
35. Jensen, T., Metayer, D.L., Thorn, T.: Verification of control flow based security properties. In: Proceedings of the IEEE Symposium on Security and Privacy, pp. 89–103 (1999)
36. Jha, S., Schwoon, S., Wang, H., Reps, T.W.: Weighted pushdown systems and trust-management systems. In: Hermanns, H., Palsberg, J. (eds.) Proc. of the Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 3920, pp. 1–26. Springer, Heidelberg (2006)
37. Kidd, N., Lal, A., Reps, T.: WALi: the weighted automata library. See http://www.cs.wisc.edu/wpis/wpds/
38. Kupferman, O.: Automata theory and model checking. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
39. Muller, D.E., Schupp, P.E.: The theory of ends, pushdown automata, and second-order logic. Theor. Comput. Sci. **37**, 51–75 (1985)
40. Müller-Olm, M., Seidl, H.: Precise interprocedural analysis through linear algebra. In: Proc. of the Symp. on Principles of Programming Languages (POPL), pp. 330–341. ACM, New York (2004)
41. Parikh, R.: On context-free languages. J. ACM **13**(4), 570–581 (1966)
42. Piterman, N., Pnueli, A.: Temporal logic and fair discrete systems. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
43. Piterman, N., Vardi, M.Y.: Global model-checking of infinite-state systems. In: Alur, R., Peled, D. (eds.) Proc. of the Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 3114, pp. 387–400. Springer, Heidelberg (2004)
44. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: Proc. of the Symp. on Principle of Programming Languages (POPL), pp. 49–61. ACM, New York (1995)
45. Reps, T., Schwoon, S., Jha, S., Melski, D.: Weighted pushdown systems and their application to interprocedural dataflow analysis. Sci. Comput. Program. **58**(1–2), 206–263 (2005). Special Issue on the Static Analysis Symposium 2003
46. Sagiv, S., Reps, T.W., Horwitz, S.: Precise interprocedural dataflow analysis with applications to constant propagation. Theor. Comput. Sci. **167**(1&2), 131–170 (1996)
47. Schwoon, S.: Model-checking pushdown systems. Ph.D. thesis, TU München (2002)

48. Seth, A.: An alternative construction in symbolic reachability analysis of second order pushdown systems. Int. J. Found. Comput. Sci. **19**(4), 983–998 (2008)
49. Sharir, M., Pnueli, A.: Two approaches to interprocedural data flow analysis. In: Program Flow Analysis: Theory and Applications, pp. 189–233. Prentice-Hall, New York (1981)
50. Song, F., Touili, T.: Efficient CTL model checking for pushdown systems. In: Katoen, J.P., König, B. (eds.) Proc. of the Intl. Conf. on Concurrency Theory (CONCUR). LNCS, vol. 6901, pp. 434–449. Springer, Heidelberg (2011)
51. Steffen, B., Claßen, A., Klein, M., Knoop, J., Margaria, T.: The fixpoint-analysis machine. In: Lee, I., Smolka, S.A. (eds.) Proc. of the Intl. Conf. on Concurrency Theory (CONCUR). LNCS, vol. 962, pp. 72–87. Springer, Heidelberg (1995)
52. Suwimonteerabuth, D., Schwoon, S., Esparza, J.: jMoped: a Java bytecode checker based on Moped. In: Halbwachs, N., Zuck, L.D. (eds.) Proc. of the Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 3440, pp. 541–545. Springer, Heidelberg (2005)
53. Walukiewicz, I.: Pushdown processes: games and model checking. In: Alur, R., Henzinger, T.A. (eds.) Proc. of the Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 1102, pp. 62–74. Springer, Heidelberg (1996)
54. Walukiewicz, I.: Model checking CTL properties of pushdown systems. In: Proc. of the Conf. on Foundations of Software Technology and Theoretical Computer Science (FSTTCS). LNCS, vol. 1974, pp. 127–138. Springer, Heidelberg (2000)
55. Walukiewicz, I.: Pushdown processes: games and model-checking. Inf. Comput. **164**(2), 234–263 (2001)

Chapter 18
Model Checking Concurrent Programs

Aarti Gupta, Vineet Kahlon, Shaz Qadeer, and Tayssir Touili

Abstract Concurrent programs are in widespread use for harnessing the computing power of multi-core hardware. However, it is very challenging to develop *correct* concurrent programs. In practice, concurrency-related bugs such as data races, deadlocks, and atomicity violations are very common. In this chapter, we describe efforts based on model-checking for automatic verification and debugging of concurrent programs. The emphasis is on core ideas for reasoning about synchronizations and communication between threads and processes, while considering all possible behaviors due to their interactions.

We start by considering model-checking based on interacting pushdown system (PDS) models. In these models, each component (thread or process) is modeled as a pushdown automaton, where the stack is used to model recursion. Model checking based on pushdown automata has a close correspondence with dataflow analysis of programs, and this has been successfully used for verification of sequential programs. However, applying these methods to a system of *interacting* pushdown automata is not straightforward. Even the basic problem of reachability is undecidable in the general case. We describe some techniques that have been proposed to get around this barrier, by restricting the patterns of synchronization and communication among components.

Although PDSs provide a natural model for concurrent programs, it is difficult to apply PDS-based model-checking techniques directly to concurrent programs in practice. In addition to the formidable decidability barrier, this is also due to the huge gap between low-level PDS models and the feature-rich high-level programming

A. Gupta (✉)
Princeton University, Princeton, NJ, USA
e-mail: aartig@cs.princeton.edu

V. Kahlon
Google Inc., New York, NY, USA

S. Qadeer
Microsoft Research, Redmond, WA, USA

T. Touili
CNRS, Paris, France

E.M. Clarke et al. (eds.), *Handbook of Model Checking*,
DOI 10.1007/978-3-319-10575-8_18

languages in which concurrent programs are written. Fortunately, the successes of model-checking on finite state systems and sequential programs have provided a wealth of useful abstractions and techniques to bridge this gap. In the last part of the chapter, we will describe verification techniques for concurrent programs that are inspired by these models. They often abstract the effects of synchronization and focus on handling the complexity of reasoning about all possible behaviors. However, they can, and should, exploit insights and results of PDS-based model-checking.

18.1 Introduction

Concurrent programming has a long and rich history, motivated by the goal of harnessing parallel hardware to derive faster performance on applications of interest. These applications traditionally arose in the areas of operating systems, embedded systems, distributed databases, and large-scale scientific applications. Since the advent of multi-core hardware platforms, concurrent programming has become ever more popular, with applications in many other areas such as multi-media processing, gaming, and mobile applications. Thus, concurrent programs have now emerged outside the domains of experts. All programmers need to be aware of concurrency, either within their applications or in the larger software systems of which their applications are a part.

Concurrent programs are very difficult to develop, as well as to debug. There is a great need for systematic verification to complement (or supplement) traditional software testing, which suffers from insufficient coverage and lack of proofs. Many verification techniques have been proposed based on theorem proving, static analysis, and model-checking. This chapter will focus mainly on model-checking, although many common ideas have influenced these techniques, as well as software testing. Indeed the seminal work on model-checking [19] was proposed for synthesizing correct synchronization skeletons for concurrent (finite state) processes with respect to temporal logic specifications in CTL (Computational Tree Logic). Since then, the core ideas in model-checking have been applied in various concurrent settings, including finite state concurrent systems and concurrent multi-threaded programs. We will focus mainly on the latter in this chapter, and finite state systems are covered well in other chapters in the Handbook.

The success of model-checking on sequential programs has also led to great interest in applying it to concurrent programs. However, concurrent programs present additional challenges on top of those in model-checking sequential programs (viz. large or infinite state spaces, tradeoffs between precision and abstraction, handling of heaps, etc.). There are subtle effects due to synchronization and communication operations between threads or processes. These operations include use of locks and semaphores for mutual exclusion, wait-notify or CCS-style pair-wise rendezvous for synchronization, and multi-cast or broadcast for communication. Furthermore, the underlying model of interleaving computations of individual components often leads to an explosion in the number of system behaviors one has to reason about.

This chapter will focus on these two main challenges: precise reasoning about synchronization and communication operations, and efficient handling of component interleavings. The ideas discussed here can be combined with other techniques described elsewhere in the Handbook. (Pointers to specific related chapters are provided at the end of this section.)

We will start by considering pushdown automata models. Pushdown Systems (PDSs) have emerged as a powerful unifying framework for static analysis of sequential programs [6, 49]. Given a sequential program, data abstractions are used to derive a finite control structure, while recursion is modeled using a stack that tracks function calls and returns. Pushdown systems provide a natural model for such abstractly interpreted structures, and have in many cases led to strictly more expressive frameworks than those provided by classical inter-procedural dataflow analysis (see [49, 61]). These results highlight (i) the deep connection between dataflow analysis and the model-checking problem for PDSs, and (ii) the usefulness of PDSs as a natural framework for modeling programs.

The success of PDS-based models on sequential programs naturally led to an interest in model-checking a system of *interacting* PDSs, where the individual program components communicate and synchronize with each other, as in concurrent programs. Again, a stack models the recursion in each PDS component, to keep track of its calling context. However, a key undecidability result [60] showed that even simple properties like reachability are undecidable for systems with just two PDSs synchronizing via CCS-style pair-wise rendezvous. In [42], the undecidability result was shown to hold also for PDSs interacting via locks. These undecidability results pose a formidable barrier to extending PDS-based static analysis to concurrent programs. Therefore, much of the work using PDS-based models focuses on how to get around this undecidability barrier: by using abstractions, restricting the patterns of synchronization/communication, and identifying fragments of temporal logic for which model-checking is decidable. We describe some key results and techniques in the main part of the chapter.

PDS-based model-checking can be viewed as providing a precise framework for reasoning about both recursion *and* synchronization in concurrent programs. With its close connection to static analysis, it provides useful theoretical insights for delineating the decidability boundaries for various analyses, and for devising suitable restrictions and abstractions. However, it has not been directly applied to concurrent programs in practice, except in small instances. This is due in large part to the undecidability barrier. At the same time, there exists a huge gap between low-level PDS models and the feature-rich high-level programming languages in which concurrent programs are developed. Additional abstractions and modeling techniques are needed to bridge this gap for practical applications, and the challenges here are similar to those in model-checking of sequential programs.

Since the core undecidability barrier in model-checking of concurrent programs stems from a combination of recursion and synchronization, common strategies in practice are to bound the recursion or abstract the synchronization. In the last part of this chapter, we will describe efforts that use such strategies. They are inspired by successful model-checking on other models: finite state concurrent systems (no recursion) on one hand, and sequential programs (no synchronization) on the other.

In the setting of models based on finite state concurrent systems, the recursion in concurrent programs is typically ignored, either by inlining procedures up to some bound, or by considering only terminating or bounded executions as in systematic testing or bounded model-checking. In the setting of models based on sequential programs, recursive procedures are handled as usual (e.g., through summaries). Here, sequential program analysis (e.g., dataflow analysis, abstract interpretation, or assume-guarantee reasoning) is lifted to a concurrent program setting that deals implicitly or explicitly with *interference*, to account for synchronization/communication with other components. In a sense, the PDS-based model-checking framework grew out of this line of work. The difference is mainly in the goals and the abstractions used—PDS-based model-checking algorithms deal precisely (or at least soundly) with synchronization/communication operations (while they may use other data-based abstractions), whereas many of the other efforts using program analysis abstract the effects of synchronization/communication (or handle only some operations), sometimes even unsoundly when the goal is to find concurrency bugs such as data races, deadlocks, or atomicity violations.

We believe that PDS-based model-checking algorithms can (and should) be judiciously combined with other program abstractions and analysis techniques to advance concurrent program verification in practice. Some examples of such work are highlighted in Sect. 18.5. Finally, we also discuss trace-based dynamic model-checking methods. These are more scalable than whole-program verification, but provide non-exhaustive coverage over program inputs.

As mentioned earlier, the material covered in this chapter is related to some other chapters in the Handbook. Specifically, the section on PDS-based model-checking is related to Chap. 17, the subsection on use of partial-order reduction on finite state models is related to Chap. 6, the subsection on techniques based on sequential program models is related to Chaps. 15 and 13, with assume-guarantee reasoning related to Chap. 12.

Organization. We start by describing notation for a concurrent system and PDS-based models in Sect. 18.2. The next two sections cover PDS-based model-checking, with Sect. 18.3 considering restricted patterns of synchronization, and Sect. 18.4 considering restrictions on communication. Finally, Sect. 18.5 describes techniques based on other models—finite state systems and sequential programs.

18.2 Concurrent System Model and Notation

We consider a general concurrent system model, comprising processes running in parallel and interacting with each other, either via:

- *Synchronization*, i.e., by jointly executing a transition, e.g., rendezvous and broadcasts, or
- *Communication*, i.e., by performing operations on shared objects.

Formally, a concurrent system is defined as a tuple of the form $(\mathscr{P}, \mathscr{O}, \mathscr{T}, \nu, s_0)$, where

- $\mathscr{P} = \{P_1, \ldots, P_n\}$ is a finite set of processes
- $\mathscr{O} = \{O_1, \ldots, O_m\}$ is a finite set of objects
- $\mathscr{T}$ is a finite set of transitions
- $\nu : \mathscr{T} \to \Sigma$ is a *labeling function*, and
- s_0 is the initial state of the system.

Each process $P_i \in \mathscr{P}$ has a finite nonempty set of *local states*, or *control locations*. Processes are pair-wise disjoint.

Processes can access a finite set of objects. An object O is defined as a pair (V, OP), where V is the set of possible values of the object and OP is the set of operations that can be performed on the object. Each operation $op_i \in OP$ is a (possibly partial) function $IN_i \times V \to OUT_i \times V$, where IN_i and OUT_i represent, respectively, the set of possible inputs and outputs of the operation.

A global state s of a concurrent system is an element of the set $S = P_1 \times \cdots \times P_n \times V_1 \times \cdots \times V_m$. A state $(s[1], \ldots, s[n], v[1], \ldots, v[m])$ assigns to each process P_i a local state $s[i] \in P_i$ and associates a value $v[j] \in V_j$ with each object O_j. The initial state $s_0 \in S$. For control location l and global state s, we use $l \in s$ to mean that $\exists i \in [1 \ldots n] : l = s[i]$.

A transition $t \in \mathscr{T}$ is a tuple (L, G, C, L'), where, intuitively speaking, L and L' are, respectively, the initial and final local control locations of processes participating in t, G is the guard expressing the condition under which t can be executed, and C is a function capturing updates on the communication objects. Formally, L and L' are nonempty subsets of $\cup_i P_i$ such that for each $i \in [1 \ldots n]$, $|L \cap P_i| = |L' \cap P_i| \leq 1$. The guard G is a conjunction of conditions c_j, where $c_j : V_1 \times \cdots \times V_m \to \{true, false\}$ is a Boolean function defined on the values of the communication objects. The command C is a function $C : V_1 \times \cdots \times V_n \to V_1 \times \cdots \times V_n$ defined as a sequential composition of operations on objects such that an operation that updates the value of O_j cannot be followed by any other operation on O_j.

A transition $t = (L, G, C, L')$ is *enabled* in a global state s if $L \subseteq s$, i.e., for each $l \in L$, $l \in s$, and G is true in s. If t is not enabled in s, it is said to be *disabled* in s. A transition t that is enabled in a state $s = (s[1], \ldots, s[n], v[1], \ldots, v[m])$ can be *executed*. As a result of the execution the system reaches a state $s' = (s'[1], \ldots, s'[n], v'[1], \ldots, v'[m])$, where

1. $\{s'[1], \ldots, s'[n]\} = (\{s[1], \ldots, s[n]\} \setminus L) \cup L'$, and
2. the command C maps $(v[1], \ldots, v[m])$ to $(v'[1], \ldots, v'[m])$.

The state s' is called the *successor* of s via t. We use $s \xrightarrow{t} s'$ to denote the fact that global state s' results from global state s via execution of transition t.

Note that this concurrent system model is general in that each process can be further modeled as a finite state process, a sequential program, or a pushdown system. We will consider the specific case of interacting pushdown systems in more detail later.

18.2.1 Synchronization and Communication

We consider the following standard primitives for communication and synchronization that can be used to model available operations in specific programming languages:

- *Locks*: Locks are used to enforce mutual exclusion. Transitions acquiring and releasing lock l are labeled with $acquire(l)$ and $release(l)$, respectively.
- *Rendezvous (Wait-Notify)*: We consider two notions of rendezvous: CCS-style *Pair-wise Rendezvous* and the more expressive *Asynchronous Rendezvous* (motivated by the `wait()` and `notify()` primitives of Java, but not identical). Pair-wise send and receive rendezvous are labeled with $a!$ and $a?$, respectively. Let $c_{11} \xrightarrow{a!} c_{12}$ and $c_{21} \xrightarrow{a?} c_{22}$ denote the pair-wise send and receive transitions of P_1 and P_2, respectively. For the pair-wise rendezvous to be enabled, both P_1 and P_2 must be in local control states c_{11} and c_{21} simultaneously, and the send and receive transitions are taken synchronously in one step. If P_1 is in c_{11} but P_2 is not in c_{12} then P_1 cannot execute the send transition, and vice versa. The asynchronous rendezvous send and receive transitions, labeled with $a\uparrow$ and $a\downarrow$, respectively, do not require this synchronous operation. The difference between pair-wise rendezvous and asynchronous rendezvous is that the send transition is blocking in the former but non-blocking in the latter. Thus a transition of the form $c_{11} \xrightarrow{a\uparrow} c_{12}$ can be executed irrespective of whether a matching receive transition of the form $c_{21} \xrightarrow{a\downarrow} c_{22}$ is currently enabled or not. On the other hand, the execution of a receive transition requires a matching send transition to be enabled, with both the send and receive then being executed synchronously.
- *Broadcasts (Notify-All)*: Broadcast send and receive rendezvous (again, motivated by the `wait()` and `notifyAll()` primitives of Java, but not identical) are labeled with $a!!$ and $a??$, respectively. If $b_{11} \xrightarrow{a!!} b_{12}$ is a broadcast send transition and $b_{21} \xrightarrow{a??} b_{22}, \ldots, b_{n1} \xrightarrow{a??} b_{n2}$ are the matching broadcast receives, then the receive transitions block pending the enabling of the send transition. The send transition, on the other hand, is non-blocking and can always be executed. Its execution is carried out synchronously with *all* the currently enabled receive transitions labeled with $a??$.

18.2.2 Specification Logic

We consider correctness properties expressed as multi-index temporal logic formulae, where in a k-index formula, the atomic propositions are interpreted over the local control states of k components. Many interesting properties can be expressed as single- or double-index properties. For example, liveness can be expressed as a single-index property, while for specifying the presence of a data race one needs a double-index property.

We use $L(Op_1, \ldots, Op_k)$, where $Op_i \in \{\mathsf{X}, \mathsf{F}, \mathsf{U}, \mathsf{G}, \overset{\infty}{\mathsf{F}}\}$, to denote the fragment of double-indexed linear temporal logic (LTL) comprising formulae of the form $\mathsf{E}f$. Here, f is an LTL formula in positive normal form (PNF), viz., only atomic propositions are negated, built using the operators $Op_1, \ldots, Op_k$ and the Boolean connectives $\vee$ and $\wedge$. Here X "next-time", F "sometimes", U, "until", G "always", and $\overset{\infty}{\mathsf{F}}$ "infinitely-often" denote the standard temporal operators and E is the "existential path quantifier." Note that $L(\mathsf{X}, \mathsf{U}, \mathsf{G})$ is full LTL.

18.2.3 Interacting Pushdown System (PDS) Model

We consider multi-threaded programs wherein threads synchronize using the standard primitives—locks, pair-wise rendezvous, asynchronous rendezvous, and broadcasts. Each thread is modeled as a *Pushdown System (PDS)* [6]. A PDS has a finite control part corresponding to the (abstract) valuations of the variables of the thread and a stack which models recursion. The stack is used only to track the context, i.e., the order in which functions are called in reaching a given control location. The properties we consider refer only to valuations in control locations, not the stack contents.

Formally, a PDS is a five-tuple $P = (Q, Act, \Gamma, c_0, \Delta)$, where Q is a finite set of *control locations*, Act is a finite set of *actions*, Γ is a finite *stack alphabet*, and $\Delta \subseteq (Q \times \Gamma) \times Act \times (Q \times \Gamma^*)$ is a finite set of *transition rules*. If $((p, \gamma), a, (p', w)) \in \Delta$ then we write $\langle p, \gamma \rangle \overset{a}{\hookrightarrow} \langle p', w \rangle$. A *configuration* of P is a pair $\langle p, w \rangle$, where $p \in Q$ denotes the control location and $w \in \Gamma^*$ the *stack content*. We call $\mathbf{c}_0$ the *initial configuration* of P. The set of all configurations of P is denoted by $\mathscr{C}$. For each action a, we define a relation $\overset{a}{\rightarrow} \subseteq \mathscr{C} \times \mathscr{C}$ as follows: if $\langle q, \gamma \rangle \overset{a}{\hookrightarrow} \langle q', w \rangle$, then $\langle q, \gamma v \rangle \overset{a}{\rightarrow} \langle q', wv \rangle$ for every $v \in \Gamma^*$.

Let $\mathscr{P}$ be a multi-PDS system comprising of the PDSs $P_1, \ldots, P_n$, where $P_i = (Q_i, Act_i, \Gamma_i, \mathbf{c}_i, \Delta_i)$. In addition to Act_i, we assume that each P_i has special action symbols labeling transitions *implementing* synchronization primitives. These synchronizing action symbols are shared commonly across all PDSs.

A concurrent program with n PDSs and m locks $l_1, \ldots, l_m$ is formally defined as a tuple of the form $\mathscr{P} = (P_1, \ldots, P_n, L_1, \ldots, L_m)$, where for each i, $P_i = (Q_i, Act_i, \Gamma_i, c_i, \Delta_i)$ is a pushdown system (thread), and for each j, $L_j \subseteq \{\bot, P_1, \ldots, P_n\}$ is the possible set of values that lock l_j can be assigned. A global configuration of $\mathscr{P}$ is a tuple $c = (t_1, \ldots, t_n, l_1, \ldots, l_m)$ where $t_1, \ldots, t_n$ are, respectively, the configurations of PDSs $P_1, \ldots, P_n$ and $l_1, \ldots, l_m$ the values of the locks. If no thread holds lock l_i in configuration c, then $l_i = \bot$, else l_i is the thread currently holding it. The initial global configuration of $\mathscr{P}$ is $(c_1, \ldots, c_n, \bot, \ldots, \bot)$, where c_i is the initial configuration of PDS P_i. Thus all locks are *free* to start with. We extend the relation $\overset{a}{\longrightarrow}$ to global configurations of $\mathscr{P}$ in the usual way.

The reachability relation $\Rightarrow *$ is the reflexive and transitive closure of the successor relation $\rightarrow$ defined above. A sequence $x = x_0, x_1, \ldots$ of global configurations of $\mathscr{P}$ is a *computation* if x_0 is the initial global configuration of $\mathscr{P}$

and for each i, $x_i \xrightarrow{a} x_{i+1}$, where either for some j, $a \in Act_j$, or for some k, $a = release(l_k)$ or $a = acquire(l_k)$ or pair-wise rendezvous send $a = b!$ or receive $a = b?$, or asynchronous rendezvous send $a = b\uparrow$ or receive $a = b\downarrow$, or broadcast send $a = b!!$ or receive $a = b??$. Given a thread T_i and a reachable global configuration $\mathbf{c} = (c_1, \ldots, c_n, l_1, \ldots, l_m)$ of $\mathscr{P}$, we use *Lock-Set*$(T_i, \mathbf{c})$ to denote the set of locks held by T_i in $\mathbf{c}$, viz., the set $\{l_j \mid l_j = T_i\}$. Also, given a thread T_i and a reachable global configuration $\mathbf{c} = (c_1, \ldots, c_n, l_1, \ldots, l_m)$ of $\mathscr{P}$, the *projection* of $\mathbf{c}$ onto T_i, denoted by $c \downarrow T_i$, is defined to be the configuration $(c_i, l'_1, \ldots, l'_m)$ of the concurrent program comprising solely the thread T_i, where $l'_i = T_i$ if $l_i = T_i$, and $\perp$ otherwise (locks not held by T_i are released).

18.3 PDS-Based Model Checking: Synchronization Patterns

Dataflow analysis for sequential programs can exploit the fact that the model-checking problem for a PDS is decidable for very expressive classes of properties—both linear and branching time (cf. [6, 74]). Analogously to the sequential case, inter-procedural dataflow analysis for concurrent multi-threaded programs can be formulated as a model-checking problem for *interacting* PDSs. However, this problem is less robustly decidable than the one for a single PDS. Indeed, a key undecidability result given in [60] showed that even simple properties like reachability are undecidable for systems with just two PDSs synchronizing via CCS-style pair-wise rendezvous. In [42], it was shown that reachability is undecidable for PDSs with arbitrary lock accesses.

A fundamental obstacle here is the undecidability of checking the non-emptiness of the intersection of two context-free languages. The implication is that if, in a system comprising of two PDSs, the coupling between them is strong enough to accept the intersection of the context-free languages accepted by these PDSs, then the model-checking problem becomes undecidable. This strong coupling can result from: (i) the synchronization primitives being sufficiently expressive, e.g., pair-wise rendezvous or broadcasts, or (ii) the property being strong enough. Thus, to ensure decidability, we need to limit the expressiveness of either the synchronization primitives or the property being model-checked.

Interestingly, in practice concurrent programs have a lot of inherent structure that can potentially be exploited. This has led to decidability results and efficient PDS-based model-checking procedures for some problems of practical interest and for various fragments of temporal logic. Another related problem is deciding *static pair-wise reachability* of two individual control states in a dual-PDS system, which provides a basic building block for various model-checking and dataflow analysis procedures.

Specifically, it has been shown that the model-checking problems for $L(\mathsf{F}, \mathsf{G})$ and $L(\mathsf{U})$ are undecidable for dual-PDS systems wherein the PDSs *do not interact at all* with each other [41]. Here, the logics are powerful enough to encode the disjointness problem for context-free languages. For the sub-logic $L(\mathsf{X}, \mathsf{G})$, model-checking is decidable for PDSs interacting via pair-wise rendezvous, asynchronous

rendezvous, or broadcasts [41]. For the sub-logic $L(\mathsf{X}, \mathsf{F}, \overset{\infty}{\mathsf{F}})$, the decidability of model-checking depends on the synchronization primitives allowed. It is decidable for nested locks [41] and for bounded lock chains [39], but is undecidable for pair-wise rendezvous, asynchronous rendezvous, or broadcasts [41].

In this section, we consider in detail some results and procedures for model-checking and pair-wise reachability problems for PDSs with restrictions on synchronization primitives—nested locks, bounded lock chains, and rendezvous. In the next section, we consider various model-checking procedures for various restrictions on the models of communication.

18.3.1 Programs with Nested Locks

Nested locks are a prime example of how programming patterns can be exploited to yield decidability of the model-checking problem for several important temporal logic fragments for interacting pushdown systems [40, 42].

We say that a concurrent program accesses locks in a nested fashion if and only if along each computation of the program a thread can only release the last lock that it acquired along that computation, and which has not yet been released. The case of nested locks is practically important as most lock usage in concurrent programs is nested. Indeed, standard programming practice guidelines typically recommend that programs use locks in a nested fashion. This is even enforced in some languages, e.g., Java (version 1.4) and C#, where locks are guaranteed to be nested.

18.3.1.1 Pair-wise Reachability for Nested Locks

We first consider the pair-wise reachability problem in a concurrent program with nested locks. For this, it is useful to consider the notion of an *acquisition history*.

Definition 1 (Acquisition History) [42] Let x be a global computation of a concurrent program $\mathscr{P}$ leading to global configuration c. Then for thread T_i and lock l_j of $\mathscr{P}$ such that $j \in \mathit{Lock\text{-}Set}(T_i, c)$, we define $AH(T_i, l_j, x)$ to be the set of locks that were acquired (and possibly released) by T_i after the last acquisition of l_j by T_i along x.

The key feature of an acquisition history is that it can be computed in a thread-local fashion, much like lock-sets. This makes possible a compositional decision procedure for reachability, and also makes it amenable in other verification settings (discussed later in Sect. 18.3.4).

Using the notion of an acquisition history, we can derive the following important result regarding decomposition of the reachability problem for nested locks.

Theorem 1 (Decomposition Result for Nested Locks) [42] *Let $\mathscr{P}$ be a concurrent program comprising two threads T_1 and T_2 with nested locks. Then two control states a_1 and b_2 of T_1 and T_2, respectively, are backward reachable from configurations $\mathbf{d}_1$ and $\mathbf{d}_2$ of T_1 and T_2 respectively if and only if there exist configurations $\mathbf{c}_1$ and $\mathbf{c}_2$ and computations x and y, from $\mathbf{c}_1$ and $\mathbf{c}_2$ to $\mathbf{d}_1$ and $\mathbf{d}_2$, respectively, such that the acquisition histories of x and y are compatible.*

The acquisition histories of x and y are compatible if there do not exist locks $l \in \text{Lock-Set}(P_1, \mathbf{d}_1)$ and $l' \in \text{Lock-Set}(P_2, \mathbf{d}_2)$ such that $l \in AH(P_2, l', y)$ and $l' \in AH(P_1, l, x)$.

Essentially, the Decomposition Result allows reduction of the problem of deciding reachability of one global configuration from another in a dual-PDS system to reachability problems for local configurations of the individual PDSs, thereby avoiding the explosion in interleavings.

18.3.1.2 Model-Checking Programs with Nested Locks

For concurrent programs with threads synchronizing via nested locks, it has been shown [40, 41] that:

- The model-checking problem is undecidable for $L(\mathsf{U})$ and $L(\mathsf{G})$. This implies that in order to get decidability for dual-PDS systems (PDS systems with two threads) interacting via nested locks, we have to restrict ourselves to the sub-logic $L(\mathsf{X}, \mathsf{F}, \overset{\infty}{\mathsf{F}})$.
- For the fragment $L(\mathsf{X}, \mathsf{F}, \overset{\infty}{\mathsf{F}})$ of LTL, the model-checking problem is decidable.

The undecidability results for model-checking $L(\mathsf{U})$ and $L(\mathsf{G})$ follow via reduction to the disjointness problem for context-free languages.

Decidability of the model-checking problem for the fragment $L(\mathsf{X}, \mathsf{F}, \overset{\infty}{\mathsf{F}})$ of LTL for concurrent programs with nested locks leverages the standard automata-theoretic approach for model-checking. Given an $L(\mathsf{X}, \mathsf{F}, \overset{\infty}{\mathsf{F}})$ formula f, we build an automaton accepting global states of the given concurrent program satisfying f. As usual, this automaton construction is defined for the basic temporal operators of $L(\mathsf{X}, \mathsf{F}, \overset{\infty}{\mathsf{F}})$, i.e., F, $\overset{\infty}{\mathsf{F}}$, and X, and the Boolean connectives $\wedge$ and $\vee$. In other words, we start by building, for each atomic proposition *prop* of f, an automaton accepting the set of states of the given concurrent program satisfying *prop*. Then, we leverage the constructions for the basic temporal operators and Boolean connectives to recursively build the automaton accepting the set of states satisfying f via an inside-out traversal of f. Finally, if the initial state of the given concurrent program is accepted by the resulting automaton, the program satisfies f.

The above approach, which is standard for LTL model-checking of finite state and pushdown systems, exploits the fact that for model-checking it suffices to reason

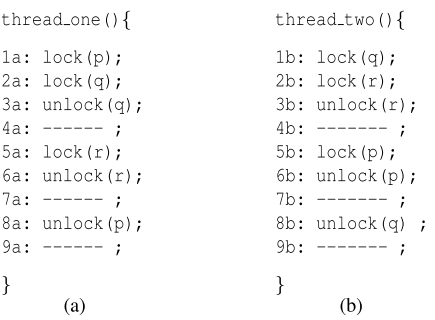


Fig. 1 Program $\mathscr{P}$ with threads P_1 (**a**) and P_2 (**b**)

about regular sets of configurations of these systems. These sets can be captured using *regular automata*, which then reduces model-checking to computing regular automata for each of the temporal operators and Boolean connectives. For general concurrent programs, the sets of configurations that we need to reason about for model-checking are *not* regular, and therefore cannot be captured via regular automata. However, for a dual-PDS system where PDSs interact via nested locks, we can represent regular sets of configurations by using the notion of a *Lock-Constrained Multi-Automata Pair (LMAP)* [40], which we briefly review next.

Lock-Constrained Multi-automata Pair (LMAP). The main motivation behind defining an LMAP is to decompose the representation of a regular set of configurations of a dual-PDS system $\mathscr{P}$ comprising PDSs P_1 and P_2 into a pair of regular sets of configurations of the individual PDSs P_1 and P_2. An LMAP accepting a set R of configurations of $\mathscr{P}$ is a pair of multi-automata $M = (M_1, M_2)$, where M_i is a multi-automaton accepting the regular set R_i of local configurations of P_i occurring in a global configuration in R. A key advantage of this decomposition is that performing operations on M, for instance computing the pre^*-closure of R, reduces to performing the same operations on the individual MAs M_i. This avoids the state explosion problem thereby making our procedure efficient.

The lock interaction among the PDSs is captured in the acceptance criterion for the LMAP via the concept of *Backward* and *Forward Acquisition Histories* [40] which we briefly describe next, followed by a formulation of the Decomposition Result.

Motivating Example for Nested Locks. Consider a concurrent program $\mathscr{P}$ comprising the two threads shown in Fig. 1. We show that reasoning about pair-wise reachability can be reduced to reasoning about reachability of control locations in the individual threads.

Observe that $\mathscr{P} \models \mathsf{EF}(\mathtt{4a} \wedge \mathtt{4b})$ but $\mathscr{P} \not\models \mathsf{EF}(\mathtt{4a} \wedge \mathtt{7b})$ even though disjoint sets of locks, viz., $\{p\}$ and $\{q\}$, are held at `4a` and `7b`, respectively. The key point is that the simultaneous reachability of two control locations of P_1 and P_2 depends not only on the lock-sets held at these locations, but also on the patterns of lock acquisition along the computation paths of $\mathscr{P}$ leading to these control locations. These patterns are captured using the notions of backward and forward acquisition histories.

Indeed, if P_1 executes first, it acquires p and does not release it along any path leading to 4a. This prevents P_2 from acquiring p, which it requires in order to transit from 1b to 7b. Similarly if P_2 executes first, it acquires q thereby preventing P_1 from transiting from 1a to 4a, which would require it to acquire and release lock q. This creates an unresolvable cyclic dependency. These dependencies can be formally captured using the notions of backward and forward acquisition histories described below.

Definition 2 (Forward Acquisition History) [41] For a lock l held by P_i at a control location d_i, the forward acquisition history of l along a local computation x_i of P_i leading from c_i to d_i, denoted by $\mathsf{fah}(P_i, c_i, l, x_i)$, is the set of locks that have been acquired (and possibly released) by P_i since the last acquisition of l by P_i in traversing forward along x_i from c_i to d_i. In case l is not acquired but held in each state along x_i then $\mathsf{fah}(P_i, c_i, l, x_i)$, is simply the set of locks that have been acquired (and possibly released) by P_i along x_i.

Observe that along any local computations x_1 and x_2 of P_1 and P_2 leading to control locations 4a and 7b, respectively, $\mathsf{fah}(P_1, \texttt{4a}, p, x_1) = \{q\}$ and $\mathsf{fah}(P_2, \texttt{7b}, q, x_2) = \{p, r\}$. Also, along any local computations x_1' and x_2' of P_1 and P_2 leading to control locations 4a and 4b, respectively, $\mathsf{fah}(P_1, \texttt{4a}, p, x_1') = \{q\}$ and $\mathsf{fah}(P_2, \texttt{4b}, q, x_2') = \{r\}$. The reason $\mathsf{EF}(\texttt{4a} \wedge \texttt{7b})$ does not hold but $\mathsf{EF}(\texttt{4a} \wedge \texttt{4b})$ does is the existence of the cyclic dependency that $p \in \mathsf{fah}(P_2, \texttt{7b}, q, x_2)$ and $q \in \mathsf{fah}(P_2, \texttt{4a}, p, x_1)$ whereas no such dependency exists for the second case.

Definition 3 (Backward Acquisition History) [41] For a lock l held by P_i at a control location c_i, the backward acquisition history of l along a local computation x_i of P_i leading from c_i to d_i, denoted by $\mathsf{bah}(P_i, c_i, l, x_i)$, is the set of locks that were released (and possibly acquired) by P_i since the last release of l by P_i in traversing backwards along x_i from d_i to c_i. In case l is not released but held in each state along x_i then $\mathsf{bah}(P_i, c_i, l, x_i)$, is simply the set of locks that have been released (and possibly acquired) by P_i along x_i.

In [40], the notions of backward and forward acquisition histories were used to decide, given two global configurations $\mathbf{c}$ and $\mathbf{d}$ of $\mathscr{P}$, whether $\mathbf{d}$ is reachable from $\mathbf{c}$. The notion of forward acquisition history was used in the case where no locks are held in $\mathbf{c}$ and that of backward acquisition history in the case where no locks are held in $\mathbf{d}$.

This is illustrated in Fig. 2 where we want to decide whether $\mathbf{c}$ is backward reachable from $\mathbf{d}$. First, we assume that all locks are free in $\mathbf{d}$ (case (i) in the figure). In that case, we track the bah of each lock. In our example, lock l, initially held at $\mathbf{c}$, is first released at $\mathbf{c}_2$. Then all locks released before the first release of l belong to the bah of l. Thus, l_1 belongs to the bah of l but l_3 does not. On the other hand, if in $\mathbf{c}$ all locks are free (case (ii) in the figure), then we track the fah of each lock. If a lock l held at $\mathbf{d}$ is last acquired at $\mathbf{c}_3$ then all locks acquired since the last acquisition of

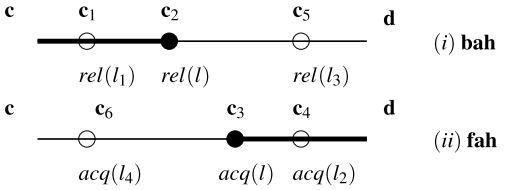


Fig. 2 Forward vs. backward acquisition history

l belong to the fah of l. Thus in our example, l_2 belongs to the forward acquisition history of l but l_4 does not.

When testing for backward reachability of $\mathbf{c}$ from $\mathbf{d}$ in $\mathscr{P}$, it suffices to test whether there exist local paths x and y of the individual PDSs from states $\mathbf{c}_1 = \mathbf{c} \downarrow P_1$ to $\mathbf{d}_1 = \mathbf{d} \downarrow P_1$ and from $\mathbf{c}_2 = \mathbf{c} \downarrow P_2$ to $\mathbf{d}_2 = \mathbf{d} \downarrow P_2$, respectively, such that along x and y lock operations can be executed in an acquisition-history-compatible fashion as formulated in the Decomposition Result below.

Theorem 2 (Decomposition Result) [41] *Let $\mathscr{P}$ be a dual-PDS system comprising the two PDSs P_1 and P_2 with nested locks. Then configuration $\mathbf{c}$ of $\mathscr{P}$ is backward reachable from configuration $\mathbf{d}$ iff configurations $\mathbf{c}_1 = \mathbf{c} \downarrow P_1$ of P_1 and $\mathbf{c}_2 = \mathbf{c} \downarrow P_2$ of P_2 are backward reachable from configurations $\mathbf{d}_1 = \mathbf{d} \downarrow P_1$ and $\mathbf{d}_2 = \mathbf{d} \downarrow P_2$, respectively, via local computation paths x and y of PDSs P_1 and P_2, respectively, such that*

1. *Lock-Set*$(P_1, \mathbf{c}_1) \cap$ *Lock-Set*$(P_2, \mathbf{c}_2) = \emptyset$,
2. *Lock-Set*$(P_1, \mathbf{d}_1) \cap$ *Lock-Set*$(P_2, \mathbf{d}_2) = \emptyset$,
3. *Locks-Acq*$(x) \cap$ *Locks-Held*$(y) = \emptyset$ *and* *Locks-Acq*$(y) \cap$ *Locks-Held*$(x) = \emptyset$, *where for path z,* *Locks-Acq*(z) *is the set of locks that are acquired (and possibly released) along z and* *Locks-Held*(z) *is the set of locks that are held in all states along z.*
4. *there do not exist locks $l \in$ Lock-Set$(P_1, \mathbf{c}_1) \backslash$Locks-Held$(x)$ and $l' \in$ Lock-Set$(P_2, \mathbf{c}_2) \backslash$Locks-Held$(y)$ such that $l \in \mathsf{bah}(P_2, \mathbf{c}_2, l', y)$ and $l' \in \mathsf{bah}(P_1, \mathbf{c}_1, l, x)$.*
5. *there do not exist locks $l \in$ Lock-Set$(P_1, \mathbf{d}_1) \backslash$Locks-Held$(x)$ and $l' \in$ Lock-Set$(P_2, \mathbf{d}_2) \backslash$Locks-Held$(y)$ such that $l \in \mathsf{fah}(P_2, \mathbf{c}_2, l', y)$ and $l' \in \mathsf{fah}(P_1, \mathbf{c}_1, l, x)$.*

Intuitively, conditions 1 and 2 ensure that the locks held by P_1 and P_2 in a global configuration of $\mathscr{P}$ must be disjoint; condition 3 ensures that if a lock held by a PDS, say P_1, is not released along the entire local computation x, then it cannot be acquired by the other PDS P_2 all along its local computation y, and vice versa; and conditions 4 and 5 ensure compatibility of the acquisition histories, viz., the absence of cyclic dependencies as discussed above.

The Decomposition Result allows us to reduce the pre^*-closure computation of a regular set of configurations of a dual-PDS system to that of its individual acquisition-history-augmented PDSs.

Towards that end, we first need to extend existing pre^*-closure computation procedures for regular sets of configurations of a single PDS to handle regular sets

of *acquisition-history-augmented* (ah-augmented) configurations. An acquisition-history-augmented configuration $\mathbf{c}_i$ of P_i is of the form $(\langle p_i, w\rangle, l_1, \ldots, l_m, \mathsf{bah}_1, \ldots, \mathsf{bah}_m, \mathsf{fah}_1, \ldots, \mathsf{fah}_m)$ where for each i, fah_i and bah_i are lock-sets storing, respectively, the forward and backward acquisition history of lock l_i. Since the procedure is similar to the ones for fah- and bah-augmented configurations given in [40], its formal description is omitted. The key result is the following.

Theorem 3 (ah-enhanced pre^*-computation) [41] *Given a PDS P, and a regular set of* ah*-augmented configurations accepted by a multi-automaton $\mathscr{A}$, we can construct a multi-automaton $\mathscr{A}_{pre^*}$ recognizing $pre^*(Conf(\mathscr{A}))$ in time polynomial in the sizes of $\mathscr{A}$ and the control states of P and exponential in the number of locks of P.*

Acceptance Criterion for LMAPs. The absence of cyclic dependencies encoded using bahs and fahs is used in the acceptance criterion for LMAPs to factor in lock interaction among the PDSs that prevents them from simultaneously reaching certain pairs of local configurations.

As mentioned earlier, we construct LMAPs for each of the temporal and Boolean operators in $L(\mathsf{X}, \mathsf{F}, \overset{\infty}{\mathsf{F}})$ for model-checking concurrent programs with nested locks (details are available in related publications [40, 41]). This leads to the following main decidability result.

Theorem 4 ($L(\mathsf{X}, \mathsf{F}, \overset{\infty}{\mathsf{F}})$-decidability) [41] *The model-checking problem for $L(\mathsf{X}, \mathsf{F}, \overset{\infty}{\mathsf{F}})$ is decidable for PDSs interacting via nested locks in time polynomial in the sets of control states of the given dual-PDS system and exponential in the number of locks.*

18.3.2 Programs with Locks: Lock Causality Graph

In general, for reasoning about reachability on programs with locks, both nested as well as non-nested, the concept of a *Lock Causality Graph* (LCG) is very useful. We describe this next.

Consider the example concurrent program $\mathscr{P}$ comprising threads T_1 and T_2 shown in Fig. 3. Suppose that we are interested in deciding whether *a6* and *b8* are simultaneously reachable. For that to happen there must exist local paths x^1 and x^2 of T_1 and T_2 leading to *a6* and *b8*, respectively, along which locks can be acquired in a consistent fashion. We start by constructing a *lock causality graph* $G_{(x^1,x^2)}$ that captures the constraints imposed by locks on the order in which statements along x^1 and x^2 need to be executed in order for T_1 and T_2 to simultaneously reach *a6* and *b8*. The nodes of this graph are (the relevant) locking/unlocking statements executed along x^1 and x^2. For statements c_1 and c_2 of $G_{(x^1,x^2)}$, there exists an edge from c_1

```
void T1(){
a1: lock(l1);
a2: lock(l2);
a3: lock(l6);
a4: unlock(l1);
a5: unlock(l2);
a6: Race0;
}
```

```
void T2(void){
b1: lock(l6);
b2: lock(l2);
b3: unlock(l6);
b4: lock(l5);
b5: unlock(l5);
b6: lock(l1);
b7: unlock(l2);
b8: Race1;
}
```

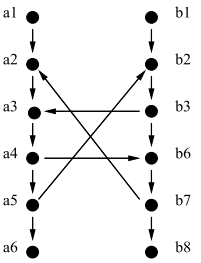


Fig. 3 An example program and lock causality graph

to c_2, denoted by $c_1 \rightsquigarrow c_2$, if c_1 must be executed before c_2 in order for T_1 and T_2 to simultaneously reach *a6* and *b8*, respectively.

Causality Constraints:

(a) Consider lock l_1 held at *b8*. Note that once T_2 acquires l_1 at location *b6*, it is not released along the path from *b6* to *b8*. Since we are interested in the pair-wise reachability of $a6$ and $b8$, T_2 cannot progress beyond location *b8* and therefore cannot release l_1. Thus we have that once T_2 acquires l_1 at *b6*, T_1 cannot acquire it thereafter. If T_1 and T_2 are to simultaneously reach *a6* and *b8*, respectively, the last transition of T_1 that releases l_1 before reaching *a6*, i.e., *a4*, must be executed before *b6*. Thus $a4 \rightsquigarrow b6$.

(b) Causal constraints can be deduced in another way. Consider the constraint $a4 \rightsquigarrow b6$. At location *b6*, lock l_2 is held which was acquired at *b2*. Also, once l_2 is acquired at *b2* it is not released till after T_2 exits *b6*. Thus if l_2 has been acquired by T_1 before reaching *a4* it must be released before *b2* (and hence *b6*) can be executed. In our example, the last statement to acquire l_2 before *a4* is *a2*. The unlock statement corresponding to *a2* is *a5*. Thus, $a5 \rightsquigarrow b2$.

Computing the Lock Causality Graph. Given finite local paths x^1 and x^2 of threads T_1 and T_2 leading to control locations c_1 and c_2, respectively, the procedure (see Algorithm 1) to compute $G_{(x^1,x^2)}$, adds the causality constraints one by one (of type (a) via steps 3–7, and of type (b) via steps 9–19) till we reach a fixpoint. Throughout the description of Algorithm 1, for $i \in [1 \ldots 2]$, we use i' to denote an integer in $[1 \ldots 2]$ other than i. Note that condition 14 in the algorithm ensures that we do not add edges representing causality constraints that can be deduced from existing edges. Also, steps 21–23 preserve the local causality constraints along x^1 and x^2. The causality graph $G_{(x^1,x^2)}$ for paths $x^1 = a1, \ldots, a6$ and $x^2 = b1, \ldots, b8$ is shown in Fig. 3.

Necessary and Sufficient Condition for Pair-wise Reachability. Let x^1 and x^2 be local computations of T_1 and T_2 leading to c_1 and c_2. Since each causality constraint in $G_{(x^1,x^2)}$ is a *happens-before* constraint, we see that in order for c_1 and c_2 to be pair-wise reachable $G_{(x^1,x^2)}$ has to be acyclic. In fact, it turns out that acyclicity is also a sufficient condition.

Algorithm 1 Computing the Lock Causality Graph

Input: Local paths x^1 and x^2 of T_1 and T_2 leading to c_1 and c_2, respectively
Initialize the vertices and edges of $G_{(x^1,x^2)}$ to $\emptyset$
for each lock l held at location c_i **do**
  if c and c' are the last statements to acquire and release l occurring along x^i and $x^{i'}$, respectively, **then**
    Add edge $c' \rightsquigarrow c$ to $G_{(x^1,x^2)}$
  end if
end for
repeat
  for each lock l **do**
    for each edge $d_{i'} \rightsquigarrow d_i$ of $G_{(x^1,x^2)}$ **do**
      Let $a_{i'}$ be the last statement to acquire l before $d_{i'}$ along $x^{i'}$ and $r_{i'}$ the matching release for $a_{i'}$
      Let r_i be the first statement to release l after d_i along x^i and a_i the matching acquire for r_i
      if l is held at either d_i or $d_{i'}$ **then**
        if there does not exist an edge $b_{i'} \rightsquigarrow b_i$ such that $r_{i'}$ lies before $b_{i'}$ along $x^{i'}$ and a_i lies after b_i along x^i **then**
          add edge $r_{i'} \rightsquigarrow a_i$ to $G_{(x^1,x^2)}$
        end if
      end if
    end for
  end for
until no new statements can be added to $G_{(x^1,x^2)}$
for $i \in [1..2]$ **do**
  Add edges between all statements of x^i occurring in $G_{(x^1,x^2)}$ to preserve their relative ordering along x^i
end for

Theorem 5 (Acyclicity) *Locations c_1 and c_2 are pair-wise reachable if there exist local paths x^1 and x^2 of T_1 and T_2 leading to c_1 and c_2, respectively, such that $G_{(x^1,x^2)}$ is acyclic.*

18.3.3 Programs with Bounded Lock Chains

While the use of nested locks remains the most popular pattern, there are niche applications where locks are non-nested and lock-chaining is required. Lock-chaining occurs when the scopes of two mutexes overlap. While one mutex is held, the code enters a region where another mutex is required. After successfully locking that second mutex, the first one is no longer needed and is released. Lock chaining is an essential device that is used for enforcing serialization. For instance, the two-phase

commit protocol that lies at the heart of serialization in databases uses lock chaining. Other classic examples where non-nested locks can occur are programs that use both mutexes and (locks associated with) wait/notify primitives (condition variables), or threads that traverse concurrent data structures, e.g., arrays, in an iterative fashion. Formally *lock chains* are defined as follows.

Definition 4 (Lock Chains) Given a computation x of a concurrent program, a lock chain of thread T is a sequence of lock acquisition statements $acq_1, \ldots, acq_n$ performed by T along x in the order listed such that for each i, the matching release of acq_i occurs after acq_{i+1} and before acq_{i+2} along x.

The *length* of a lock chain is defined to be the number of lock acquisition statements occurring along it. We say that a concurrent program $\mathscr{P}$ uses bounded lock chains if there exists B such that the length of each lock chain of a thread T of $\mathscr{P}$ occurring along a computation x is bounded by B. Note that we do not insist that all lock usage should be in the form of bounded lock chains. Rather what we require is that any lock chain induced by a computation of $\mathscr{P}$, irrespective of the locking pattern used, be bounded in length. The bounded lock chain pattern covers most cases of practical interest. It is worth pointing out that nested locks are a special case as they form lock chains of length one.

The model-checking procedure for PDSs interacting via bounded lock chains for the LTL fragment $L(\mathsf{X}, \mathsf{F}, \overset{\infty}{\mathsf{F}})$ is similar to the one for PDSs with nested locks. Given an $L(\mathsf{X}, \mathsf{F}, \overset{\infty}{\mathsf{F}})$ formula f, we build automata accepting global states of the given concurrent program satisfying f. Towards that end, we first construct automata for the basic temporal operators F, $\overset{\infty}{\mathsf{F}}$, and X, and the Boolean connectives $\wedge$ and $\vee$. Leveraging the constructions for the basic temporal operators and Boolean connectives, we can then recursively build the automaton accepting the set of states satisfying f via an inside-out traversal of f. For reasoning about PDSs with bounded lock chains we use the notion of Lock Causality Automata (LCAs) [39], which are generalizations of LMAPs used for handling nested locks. As for nested locks, the constructions of LCAs for the various temporal operators depend upon computing an LCA accepting the pre^*-closure of the set of states accepted by a given LCA. This in turn, hinges on deciding a set of *static reachability* queries between local control states of PDSs as defined below.

Definition 5 (Static Reachability for Locks) A global state (c_1, c_2) is statically reachable for locks, if there exist local paths x^1 and x^2 leading to control locations c_1 and c_2 in threads T_1 and T_2, respectively, and there exists an interleaving x of x^1 and x^2 respecting only the scheduling constraints imposed by statements using lock primitives in P (and ignoring constraints arising from data values and other synchronization primitives).

Outline of Strategy for Deciding Static Reachability. In order to decide static reachability we exploit a *small model* property. Let c_1 and c_2 be pair-wise reachable

via a global computation x of $\mathscr{P}$. If x^i is the local computation of T_i along x then by treating x^i as a sequential computation and ignoring lock interactions with the other thread, we can construct a small model y^i from x^i leading to c_i. However, naively applying the (sequential) small model property and ignoring lock interactions does not guarantee that the resulting y^is can be interleaved to form a valid global computation of $\mathscr{P}$ leading to (c_1, c_2). In order to ensure that, during the construction of y^i we need to preserve key locking/unlocking statements occurring along x^i that impact reachability of (c_1, c_2). These statements can be precisely identified via the use of a *Lock Causality Graph*. With the pair of local computations x^1 and x^2 of threads T_1 and T_2, leading to control locations c_1 and c_2, respectively, we associate the lock causality graph $G_{(x_1,x_2)}$ having the useful property that (c_1, c_2) is pair-wise reachable via a global computation resulting from an interleaving of x^1 and x^2 if and only if $G_{(x^1,x^2)}$ is acyclic (see Sect. 18.3.2). Our strategy is to exploit the sequential small model property to produce a small model y^i for x^i while ensuring that $G_{(y^1,y^2)}$ is acyclic.

In fact, our small model property works by preserving the lock causality graph, i.e., while constructing y^i from x^i we preserve all statements of $G_{(x^1,x^2)}$ occurring along x^i. Then since $G_{(y^1,y^2)}$ is the same as $G_{(x^1,x^2)}$, we have that $G_{(y^1,y^2)}$ is acyclic and so by the acyclicity result y^1 and y^2 can be interleaved to form a global computation of $\mathscr{P}$ leading to (c_1, c_2). If the size of the lock causality graph $G_{(x^1,x^2)}$ was unbounded then it would be hard to construct the desired small models y^i of bounded size. Thus it is crucial that the lock causality graph induced by x^1 and x^2 be bounded in size, which follows from the following key result.

Theorem 6 (Bounded Lock Causality Graph) [38] *If the length of each lock causality sequence generated by local paths x^1 and x^2 of threads T_1 and T_2, respectively, is bounded by B, then $G_{(x^1,x^2)}$ has at most $|L|^{B+1}$ edges, where $|L|$ is the number of locks in $\mathscr{P}$.*

This leads to the desired small model property for static reachability.

Theorem 7 (Concurrent Small Model Property) [38] *Let c_1 and c_2 be pair-wise reachable control locations of PDSs T_1 and T_2, respectively, in concurrent program $\mathscr{P}$. Then if the length of each lock chain in T_1 and T_2 is bounded by b there exists a bound B such that there is a computation of $\mathscr{P}$ of length at most B leading to a global state with T_1 and T_2 in control states c_1 and c_2, respectively. Moreover, B is a function of b and the number of locks in $\mathscr{P}$.*

A key implication of the above result is that the problem of deciding static reachability for locks reduces to a reachability problem for a finite state system. This immediately yields decidability of static pair-wise reachability for PDSs synchronizing via bounded lock chains.

18.3.4 Discussion: Lock Patterns

While the model-checking algorithms described for nested locks (Sect. 18.3.1.2) depend on computing and representing regular configurations of multi-PDS systems, note that the decomposition results essentially characterize these in terms of compatible (forward and backward) reachable computations in individual PDSs, augmented by information related to (forward and backward) lock acquisition histories. The compatibility conditions ensure that the semantics of locks (enforcing mutual exclusion) are satisfied on the global computations, without explicitly considering the interleavings. The same remark holds for the simpler case of deciding pair-wise reachability for nested locks. The ideas of lock acquisition histories have been used to develop new decision procedures for detecting atomicity violations for concurrent processes [46] and for reasoning about dynamic pushdown networks with well-nested locks [51].

In general, efficient reasoning about threads synchronizing via locks hinges on accurately and succinctly capturing the interactions between the threads resulting from lock-induced mutual exclusion constraints. While lock interactions between threads can be quite complex, not all of them may be relevant for reasoning about the property at hand. Indeed, so as not to kill parallelism among threads, locks are typically used in a very localized fashion. In this case, most of the lock interactions that one cares about would be local to the program states of interest.

The key concept of a Lock Causality Graph (LCG, Sect. 18.3.2) exposes precisely those lock interactions that are pertinent to analyzing the property at hand. It has proven to be very useful in reasoning about threads interacting via locks, and is applicable to programs with nested as well as non-nested locks. It captures mutual exclusion constraints imposed by locks in terms of causality, i.e., happens before, edges [52]. Roughly speaking, the size of an LCG as measured by the number of causality edges is a good measure of the complexity of interactions between threads inducing the LCG. In general, the size of an LCG could be unbounded. In fact, the undecidability result for threads interacting via locks [42] involves constructing threads with lock chains of infinite length that induce infinite-sized LCGs. However, for programs with bounded lock chains (Sect. 18.3.3) it is shown that the sizes of LCGs are bounded. This formalizes the intuition that one can reason efficiently about threads as long as the interaction between them is limited. In the extreme case of lock chains of length one, i.e., nested locks, LCGs reduce to acquisition histories.

The notions of lock acquisition histories and lock causality graphs are applicable beyond the settings of PDS-based model-checking. Indeed, they do not depend on PDS-specific machinery (regular configurations, or stack information). They can be used in combination with other program abstractions to enable precise reasoning about lock operations. An example of such a setting is the use of abstract domains for static analysis of concurrent programs [43]. Here, lock-based reasoning is used to eliminate global states that are unreachable, as part of an iterative procedure that applies invariant generation techniques on the resulting transaction graph. In the absence of precise reasoning about locks, the derived invariants may be too coarse to be useful.

Another example is the use of acquisition histories for tracking nested locks in trace-based predictive analysis [25]. Here, alternate schedules with bugs are predicted from the observed traces, and acquisition histories are used to rule out infeasible interleavings. Similarly, a generalization of the lock causality graph, called a universal causality graph (UCG) [44], has also been used for predictive analysis. Precise reasoning about locks helps to reduce the number of false alarms reported by predictive analysis, while improving the interleaving coverage to consider different orders on acquiring locks. Without such reasoning, other predictive techniques either ignore lock/unlock operations, or regard them as happens-before operations in the observed trace, i.e., they do not consider alternate orders on acquiring locks. These applications are highlighted in the last section of the chapter.

Finally, a different pattern called *contextual locking* has been identified for programs with locks. In contextual locking, locks are released in the same context where they are acquired, and every lock acquired in a procedure is released before the procedure returns. For programs with contextual locking, the problem of pair-wise reachability is shown to be polynomial-time decidable [15]. Note that this allows unbounded lock accesses, unlike the bounded lock chains described earlier. It is also shown that the problem becomes undecidable for re-entrant locks, i.e., where the same thread can acquire a re-entrant lock multiple times [4].

18.3.5 Programs with Rendezvous

In this section we consider programs where processes communicate via rendezvous. The reachability problem being undecidable for such programs, analysis techniques based on the computation of upper-approximations of the set of possible program paths have been proposed.

A network of pushdown systems communicating via rendezvous is called a *Communicating Pushdown System* (CPDS in short) in [8, 9, 16]. In these works, the reachability problem for CPDSs is reduced to deciding the emptiness question for the intersection of two context-free languages as follows: Let (P_1, P_2) be a CPDS, and let $C_1 \times C_2$ and $C_1' \times C_2'$ be two sets of global configurations of the system. $C_1' \times C_2'$ is reachable from $C_1 \times C_2$ if and only if there exists at least one sequence of synchronization actions that simultaneously leads P_1 from a configuration in C_1 to a configuration in C_1' and P_2 from a configuration in C_2 to a configuration in C_2'. This holds iff $L(C_1, C_1') \cap L(C_2, C_2') \neq \emptyset$, where $L(C_i, C_i')$ is the context-free language consisting of all the sequences of actions that lead P_i from C_i to C_i'.

Because deciding the emptiness of two context-free languages is undecidable, a semi-decision procedure [16] is used that, in case of termination, answers *exactly* whether the intersection is empty or not. Moreover, if $L(C_1, C_1') \cap L(C_2, C_2') \neq \emptyset$, the semi-decision procedure is *guaranteed to terminate* and return a sequence in the intersection.

The semi-decision procedure is based on a CounterExample Guided Abstraction Refinement (CEGAR) scheme as follows.

1. **Abstraction:** Compute whether an over-approximation A_i of the path language $L(C_i, C_i')$.
2. **Verification:** Check $A_1 \cap A_2 = \emptyset$, and, if so, conclude that $L(C_1, C_1') \cap L(C_2, C_2') = \emptyset$, i.e., that $C_1' \times C_2'$ is unreachable from $C_1 \times C_2$. Otherwise, compute the "counterexample" $I = A_1 \cap A_2$.
3. **Counterexample Validation:** Check whether I contains a sequence x that is in $L(C_1, C_1') \cap L(C_2, C_2')$. In this case I is not spurious: conclude that $L(C_1, C_1') \cap L(C_2, C_2') \neq \emptyset$, i.e., that $C_1' \times C_2'$ is reachable from $C_1 \times C_2$. Otherwise, proceed to the next step.
4. **Refinement:** If I is spurious, refine the over-approximations A_1 and A_2, i.e., compute other over-approximations A_1' and A_2' such that $L(C_i, C_i') \subseteq A_i' \subseteq A_i$. Then continue from step 2.

In the remainder of this section, we discuss these steps in detail. We fix two sets of global configurations $C_1 \times C_2$ and $C_1' \times C_2'$. For the sake of simplicity, we denote $L(C_1, C_1')$ by L_1, and $L(C_2, C_2')$ by L_2.

18.3.5.1 Computing Over-approximations of Path Languages

Consider an abstract lattice $(D, \leq, \sqcap, \sqcup, \bot, \top)$ associated with an idempotent semiring $(D, \oplus, \odot, \bar{0}, \bar{1})$ such that $\oplus = \sqcup$ is an associative, commutative, and idempotent $(a \oplus a = a)$ operation; $\odot$ is an associative operation; $\bar{0} = \bot$; $\bar{0}$ and $\bar{1}$ are neutral elements for $\oplus$ and $\odot$, respectively; $\bar{0}$ is an annihilator for $\odot$ $(a \odot \bar{0} = \bar{0} \odot a = \bar{0})$; and $\odot$ distributes over $\oplus$. Finally, $\leq$ is such that $x \leq x \oplus a$.

D is related to the concrete domain 2^{Lab^*} as follows:

- It contains an element v_a for every letter $a \in Lab$,
- It is associated with an abstraction function $\alpha : 2^{Lab^*} \to D$ and a concretization function $\gamma : D \to 2^{Lab^*}$ defined as follows:

$$\alpha(L) = \bigoplus_{a_1 \cdots a_n \in L} v_{a_1} \odot \cdots \odot v_{a_n}$$

and

$$\gamma(x) = \left\{ a_1 \cdots a_n \in Lab^* \;\middle|\; v_{a_1} \odot \cdots \odot v_{a_n} \leq x \right\}.$$

It is easy to see that for every language $L \subseteq Lab^*$; $\alpha(L) \in D$, and $\gamma(\alpha(L)) \supseteq L$. In other words, $\gamma(\alpha(L))$ is an over-approximation of L that is finitely represented in the abstract domain D by the element $\alpha(L)$. Intuitively, the abstract operations $\odot$ and $\oplus$ correspond to concatenation and union, respectively; $\leq$ and $\sqcap$ correspond to inclusion and intersection, respectively; and the abstract elements $\bar{0}$ and $\bar{1}$ correspond to the empty language and $\{\epsilon\}$, respectively.

Therefore, to compute the over-approximation $\gamma(\alpha(L_i))$, we need to compute its representative $\alpha(L_i)$ in the abstract domain D. Let a *finite-chain abstraction* be an abstraction such that D does not contain an infinite ascending chain, and let h be the maximal height of a chain in D. Then we have the following.

Theorem 8 ([9, 61]) *Let $P = (Q, Act, \Gamma, \Delta)$ be a PDS and C, C' be two regular sets of configurations of P, and let α be a finite-chain abstraction defined on the abstract domain D. Then $\alpha(L(C, C'))$ can be effectively computed in D in $O(h|\Delta||Q|^2)$ time.*

To check the emptiness of the intersection of the over-approximations $\gamma(\alpha(L_1))$ and $\gamma(\alpha(L_2))$, it suffices to check whether $\alpha(L_1) \sqcap \alpha(L_2) = \bot$. Indeed, using the fact that $\alpha(\emptyset) = \bot$ and $\gamma(\bot) = \emptyset$, it can be shown that

$$\forall L_1, L_2 \in Lab^*, \alpha(L_1) \sqcap \alpha(L_2) = \bot \Leftrightarrow \gamma\big(\alpha(L_1)\big) \cap \gamma\big(\alpha(L_2)\big) = \emptyset.$$

18.3.5.2 Defining Refinable Finite-Chain Abstractions

To be able to apply our CEGAR scheme, we need to define refinable finite-chain abstractions, i.e., a series $(\alpha_i)_{i \geq 1}$ such that α_i is more precise than α_j if $i > j$; i.e., for every language $L \subseteq Lab^*$, if $i > j$ then

$$L \subseteq \gamma_i\big(\alpha_i(L)\big) \subseteq \gamma_j\big(\alpha_j(L)\big).$$

For this we define the *ith-prefix abstraction* as follows: Let W_i be the set of words of Lab^* of length less than or equal to i. The abstract lattice D_i is equal to 2^{W_i}; for every $a \in Lab$, $v_a = a$; $\oplus = \cup$; $\sqcap = \cap$; $U \odot V = \{(uv)_i \mid u \in U, v \in V\}$, where $(w)_i$ is the prefix of w of length i; $\bar{0} = \emptyset$; $\bar{1} = \{\epsilon\}$; $\leq = \subseteq$.

Let α_i and γ_i be the abstraction and concretization functions associated with this domain. It is easy to see that $\alpha_i(L)$ is the set of words of L of length less than i, union the set of prefixes of length i of L, i.e., $\alpha_i(L) = \{w \mid |w| < i \text{ and } w \in L, \text{ or } |w| = i \text{ and } \exists v \in Lab^* \text{ s.t. } wv \in L\}$. Therefore, $\gamma_i(\alpha_i(L)) = \{w \in \alpha_i(L) \mid |w| < i\} \cup \{wv \mid w \in \alpha_i(L), |w| = i, v \in Lab^*\}$.

Observe that it is possible to decide whether $\alpha_i(L_1) \cap \alpha_i(L_2) = \emptyset$ because for every $L \subseteq Lab^*$, $\alpha_i(L)$ is a finite set of words.

It is easy to see that if $i > j$, then α_i is more precise than α_j. Indeed, we have

$$L \subseteq \gamma_i\big(\alpha_i(L)\big) \subseteq \gamma_j\big(\alpha_j(L)\big).$$

We have thus defined a series of refinable finite-chain abstractions $\alpha_1, \alpha_2, \alpha_3, \ldots$.

Remark 1 The *ith-prefix abstraction* is only one abstraction that can be used to instantiate the framework. Others are possible, such as the *ith-suffix* or the *ith-subword* abstractions (defined in an analogous way).

18.3.5.3 Checking Whether the Counterexample Is Spurious

It remains to check whether $I = \gamma_i(\alpha_i(L_1)) \cap \gamma_i(\alpha_i(L_2))$ contains an element x such that $x \in L_1 \cap L_2$. This amounts to deciding whether $I \cap L_1 \cap L_2 = \emptyset$. Unfortunately, this problem is undecidable because I is a regular language (because for $L \subseteq Lab^*$, $\gamma_i(\alpha_i(L))$ is regular). To sidestep this problem, we check instead whether L_1 and L_2 have a common word of length at most i. This amounts to checking whether

$$\big(\alpha_i(L_1) \cap L_1\big) \cap \big(\alpha_i(L_2) \cap L_2\big) = \emptyset.$$

This is decidable because $\alpha_i(L)$ is a finite set.

18.3.5.4 The Semi-decision Procedure

Summarizing the previous discussion, we obtain the following semi-decision procedure:

1. Initially, $i = 1$.
2. Compute the common words of length less than i, and the common prefixes of length i of $L(C_1, C_1')$ and $L(C_2, C_2')$: $I' = \alpha_i(L(C_1, C_1')) \cap \alpha_i(L(C_2, C_2'))$.
3. If $I' = \emptyset$, conclude that $L(C_1, C_1') \cap L(C_2, C_2') = \emptyset$, and that $C_1' \times C_2'$ is unreachable from $C_1 \times C_2$. Otherwise, determine whether or not I' is spurious: Check whether $I' \cap L(C_1, C_1') \cap L(C_2, C_2') \neq \emptyset$. If this holds, conclude that $L(C_1, C_1')$ and $L(C_2, C_2')$ have a common word of length less than or equal to i, and therefore, that $L(C_1, C_1') \cap L(C_2, C_2') \neq \emptyset$, and $C_1' \times C_2'$ is reachable from $C_1 \times C_2$.
4. Otherwise, increment i and proceed from step 2.

It is easy to see the following.

Theorem 9 *If $L(C_1, C_1') \cap L(C_2, C_2') \neq \emptyset$, then the above semi-decision procedure terminates with the exact solution.*

18.4 PDS-Based Model-Checking: Communication Patterns

We now consider asynchronous communication with shared memory. We define a model, called PDN, which is a collection of pushdown processes and an *observation relation* R between these processes. A process P is able to observe the control state of a process Q if P is related to Q in R. Intuitively, the control state of a process represents the memory store it owns (i.e., that the process can access in both read and write modes), and the observation relation defines its rights to read the memory stores owned by the other processes in the network. An operation of a process can be constrained by its observations, but the observed processes are never constrained by the observer.

18.4.1 Reasoning About Programs with State Observation

Formally, a Push Down Network (PDN for short) is given by a tuple $N = (P_1, \ldots, P_n, R)$ where $R \subseteq \{(i, j) \mid 1 \leq i, j \leq n, i \neq j\}$ is a binary relation defining the communication structure of the network (R defines a directed graph whose nodes are $1, \ldots, n$), and for every $i \in \{1, \ldots, n\}$, $P_i = (Q_i, \Gamma_i, \Delta_i)$ is a communicating pushdown system such that Q_i is a finite set of control states, Γ_i is a finite stack alphabet, and Δ_i is a set of transition rules of the form: $\phi : (p, \gamma) \hookrightarrow (p', w)$ where $p, p' \in Q_i$ are two control states, $\gamma \in \Gamma_i$ is the symbol popped from the stack, $w \in \Gamma_i^*$ is the string pushed onto the stack, and $\phi \subseteq \bigcup_{(i,j)\in R} Q_j$ is a set of constraints over the current control states of the other observed processes.

A *local configuration* of a process in the network, say P_i, is a word $p_i w_i \in Q_i \Gamma_i^*$ where p_i is a state and w_i is a stack content. A *configuration* of the network N is a vector $(p_1 w_1, \ldots, p_n w_n) \in \prod_{i=1}^{n} Q_i \Gamma_i^*$, where $p_i w_i$ is the local configuration of P_i.

We define a *transition relation* $\Longrightarrow_N$ between configurations such that we have $(p_1 w_1, \ldots, p_n w_n) \Longrightarrow_N (p'_1 w'_1, \ldots, p'_n w'_n)$ if and only if there is an index $i \in \{1, \ldots, n\}$ such that

- there is a rule $\phi : (p, \gamma) \hookrightarrow (p', w) \in \Delta_i$ and there exists a word $u \in \Gamma_i^*$ such that $p_i = p$, $p'_i = p'$, $w_i = \gamma u$, $w'_i = wu$, and for every $j \in \{1, \ldots, n\}$, if $(i, j) \in R$ then $p_j \in \phi$.
- $\forall j \in \{1, \ldots, n\}.\, i \neq j.\, p_j = p'_j$ and $w_j = w'_j$.

Let $\Longrightarrow_N^*$ denote the reflexive transitive closure of $\Longrightarrow_N$. Given a configuration c, the set of immediate successors of c is $post_N(c) = \{c' \in \prod_{i=1}^{n} Q_i \Gamma_i^* : c \Longrightarrow_N c'\}$. This notation can be generalized straightforwardly to sets of configurations. Let $post_N^*$ denote the reflexive transitive closure of $post_N$.

Intuitively, a network $N = (P_1, \ldots, P_n, R)$ can be seen as a collection of "standard" pushdown systems that observe each other according to the structure R: $(i, j) \in R$ means that process P_i observes (reads) the states of process P_j. If a rule $\phi : (p, \gamma) \hookrightarrow (p', w)$ is in Δ_i, then process P_i can apply the "standard" pushdown rule $(p, \gamma) \hookrightarrow (p', w)$ iff every pushdown system P_j for j s.t. $(i, j) \in R$ is in a state $p_j \in \phi \cap Q_j$. The network is in the configuration $(p_1 w_1, \ldots, p_n w_n)$ when each pushdown system P_i is in configuration $p_i w_i$.

A network $N = (P_1, \ldots, P_n, R)$ is *acyclic* (resp. *cyclic*) if the graph of its relation R is acyclic (resp. cyclic). A network consisting of a single process $N = (P, \emptyset)$ will simply be denoted by P and corresponds to the standard pushdown system P.

18.4.1.1 Symbolic Representation of PDN Configurations

Let $N = (P_1, \ldots, P_n, R)$ be a PDN where $P_i = (Q_i, \Gamma_i, \Delta_i)$. Since a configuration of N can be seen as a word of dimension n in $Q_1 \Gamma_1^* \times \cdots \times Q_n \Gamma_n^*$, a natural way to represent *infinite* sets of PDN configurations is to consider *recognizable* languages, i.e., languages that can be described as a finite union of products of n

regular languages (i.e., $L = \bigcup_{j=1}^{m} L(A_1^j) \times \cdots \times L(A_n^j)$ for some $m \in \mathbb{N}$, where A_i^j is a finite state automaton over Σ_i).

18.4.1.2 Reachability Analysis of PDNs

The reachability problem for PDNs is undecidable. It becomes decidable for acyclic networks.

Theorem 10 ([2]) *The reachability problem is decidable for acyclic PDNs. However, acyclic PDNs do not preserve recognizability.*

Definition 6 Let $N = (P_1, \ldots, P_n, R)$ be an acyclic PDN where for every $i \in \{1, \ldots, n\}$, $P_i = (Q_i, \Gamma_i, \Delta_i)$. For $i \in \{1, \ldots, n\}$, let ρ_i be a binary relation in $Q_i \times Q_i$ defined by $(p, p') \in \rho_i$ iff there exists in Δ_i a rule of the form $\phi : (p, \gamma) \hookrightarrow (p', w)$. Let ρ_i^* be the reflexive transitive closure of ρ_i.

N is *stable* iff whenever $(i, j) \in R$, then for every $p, p' \in Q_j$, if $(p, p') \in \rho_j^*$ and $(p', p) \in \rho_j^*$, then for every rule $\phi : (q, \gamma) \hookrightarrow (q', w)$ in Δ_i, $p \in \phi$ iff $p' \in \phi$.

Intuitively, N is *stable* iff whenever P_j can go from a state p to a state p' and then back to p, for some index $j \in \{1, \ldots, n\}$; we have that if $(i, j) \in R$ then the rules of Δ_i do not distinguish between the states p and p'.

Theorem 11 ([3, 69]) *Let $N = (P_1, \ldots, P_n, R)$ be a stable acyclic PDN and C be a recognizable set of configurations. Then, $post_N^*(C)$ is an effectively recognizable set.*

The construction underlying this theorem is based on the iterative application of the standard $post^*$ algorithm for standard pushdown systems [6, 23] for each pushdown component in the network. The stability of the network guarantees the termination of the iterative procedure.

18.4.2 Multiphase Acyclic Pushdown Networks

A *Multiphase Acyclic Pushdown Network* (MAPN) [3, 69] is given by a tuple $M = (N_1, \ldots, N_m, T)$ where for every $j \in \{1, \ldots, m\}$, $N_j = (P_1^j, \ldots, P_n^j, R_j)$ is an acyclic PDN where for $i \in \{1, \ldots, n\}$, $P_i^j = (Q_i, \Gamma_i, \Delta_i^j)$. T is a set of transitions of the form (N_i, Φ, N_j) where $i, j \in \{1, \ldots, m\}$ and $\Phi \subseteq \prod_{k \leq n} Q_k \Gamma_k^*$ is a recognizable set of configurations.

We can think of the network N_j as an acyclic network over the processes $(P_1, \ldots, P_n)$, where each process P_i ($i \in \{1, \ldots, n\}$) executes only the rules Δ_i^j, and where these processes observe each other according to the structure R_j. T is a

phase graph: a transition $(N_i, \Phi, N_j) \in T$ means that if the acyclic PDN N_i is in a configuration $(p_1 w_1, \ldots, p_n w_n) \in \Phi$, then the network can move from a phase where the processes behave according to the network N_i to a phase where they behave according to N_j, i.e., from N_i to N_j.

Let $\mathcal{G}$ be the underlying graph of T, i.e., $(i, j) \in \mathcal{G}$ iff there exists in T a transition of the form (N_i, Φ, N_j). We say that T is cyclic (resp. acyclic) iff $\mathcal{G}$ is cyclic (resp. acyclic). The network M is said to be cyclic (resp. acyclic) iff T is cyclic (resp. acyclic). In other words, a network is acyclic iff when the processes stop behaving according to a network N_i and start behaving like another network N_j with $j \neq i$, they will never behave like N_i again (there is no cycle involving N_i in T).

An *indexed configuration* of the MAPN is a pair $\langle (p_1 w_1, \ldots, p_n w_n), i \rangle$ where $(p_1 w_1, \ldots, p_n w_n) \in \prod_{k=1}^{n} Q_k \Gamma_k^*$, and $i \in \{1, \ldots, m\}$. The index i records the current phase of the network. A *configuration* of the MAPN is a tuple $(p_1 w_1, \ldots, p_n w_n) \in \prod_{k=1}^{n} Q_k \Gamma_k^*$.

We define a *transition relation* $\Rightarrow_M$ between indexed configurations as follows: $\langle (p_1 w_1, \ldots, p_n w_n), i \rangle \Rightarrow_M \langle (p'_1 w'_1, \ldots, p'_n w'_n), j \rangle$ if and only if:

- $(p_1 w_1, \ldots, p_n w_n) = (p'_1 w'_1, \ldots, p'_n w'_n)$, and there is $(N_i, \Phi, N_j) \in T$ such that $(p_1 w_1, \ldots, p_n w_n) \in \Phi$, or
- $(p_1 w_1, \ldots, p_n w_n) \Longrightarrow_{N_j} (p'_1 w'_1, \ldots, p'_n w'_n)$ and $i = j$.

We extend $\Rightarrow_M$ to configurations in $\prod_{k=1}^{n} Q_k \Gamma_k^*$ as follows: $(p_1 w_1, \ldots, p_n w_n) \Rightarrow_M (p'_1 w'_1, \ldots, p'_n w'_n)$ iff there exist two phase indices i and j in $\{1, \ldots, m\}$ such that $\langle (p_1 w_1, \ldots, p_n w_n), i \rangle \Rightarrow_M \langle (p'_1 w'_1, \ldots, p'_n w'_n), j \rangle$. Let $\Rightarrow_M^*$ denote the reflexive transitive closure of $\Rightarrow_M$. Let C be a set of (indexed) configurations. We define $post_M(C)$ and $post_M^*(C)$ in the usual manner. Let C be a set of indexed configurations. C is said to be recognizable if and only if the set $C_j = \{(p_1 w_1, \ldots, p_n w_n) | \langle (p_1 w_1, \ldots, p_n w_n), j \rangle \in C\}$ is recognizable for every j, $1 \leq j \leq m$. As usual, the reachability problem between two sets of (indexed) configurations C_1 and C_2, for a MAPN M, is to determine whether there are two (indexed) configurations $c_1 \in C_1$ and $c_2 \in C_2$ such that $c_1 \Rightarrow_M^* c_2$.

18.4.2.1 The Reachability Problem for MAPNs

Theorem 12 ([3, 69]) *The reachability problem is undecidable for MAPNs.*

Unfortunately, we can show that this undecidability holds even for acyclic MAPNs. It was shown in [3, 69] how solving this problem would imply a decision procedure for the Post Correspondence Problem (PCP).

Theorem 13 ([3, 69]) *The reachability problem between two (indexed) configurations is undecidable for acyclic MAPNs. This holds even if the phase graph has a single transition.*

Fortunately, reachability becomes decidable for MAPNs when the constraints in the phase graph are finite sets of configurations.

Definition 7 An MAPN $M = (N_1, \ldots, N_m, T)$ is called *finitely constrained* if T is a set of transitions of the form (N_i, Φ, N_j) where $i, j \in \{1, \ldots, m\}$ and $\Phi \subseteq \prod_{k \leq n} Q_k \Gamma_k^*$ is a *finite* set of configurations.

Proposition 1 ([3, 69]) *The reachability problem between recognizable sets of* (*indexed*) *configurations is decidable for finitely constrained MAPNs.*

Definition 8 An MAPN $M = (N_1, \ldots, N_m, T)$ is *stable* if for every $j \in \{1, \ldots, m\}$, $N_j = (P_1^j, \ldots, P_n^j, R_j)$ is a stable acyclic PDN.

Stable acyclic MAPNs effectively preserve recognizability. This is due to the fact that (1) stable acyclic PDNs effectively preserve recognizability, and (2) the phase graphs of acyclic MAPNs are acyclic. This allows us to obtain the reachability set for *stable* acyclic MAPNs by successively applying the algorithm underlying Theorem 11 a finite number of times.

Theorem 14 ([3, 69]) *Let $M = (N_1, \ldots, N_m, T)$ be a* stable *acyclic MAPN and let C be a recognizable set of* (*indexed*) *configurations of M. Then $post_M^*(C)$ is effectively recognizable.*

Since recognizable sets are effectively closed under intersection, we get the following.

Corollary 1 *The reachability problem between recognizable sets of* (*indexed*) *configurations is decidable for stable acyclic MAPNs.*

18.4.2.2 For MAPNs

Definition 9 Let $M = (N_1, \ldots, N_m, T)$ be a MAPN where for every $j \in \{1, \ldots, m\}$, $N_j = (P_1^j, \ldots, P_n^j, R_j)$ is an acyclic PDN. We define the k-switch transition relation between indexed configurations inductively as follows:

- $\langle (p_1 w_1, \ldots, p_n w_n), i \rangle \stackrel{0}{\Longrightarrow}_M \langle (p'_1 w'_1, \ldots, p'_n w'_n), j \rangle$ if and only if $i = j$ and $(p_1 w_1, \ldots, p_n w_n) \Longrightarrow_{N_i}^* (p'_1 w'_1, \ldots, p'_n w'_n)$.
- $\langle (p_1 w_1, \ldots, p_n w_n), i \rangle \stackrel{k+1}{\Longrightarrow}_M \langle (p'_1 w'_1, \ldots, p'_n w'_n), j \rangle$ if and only if there is an indexed configuration $\langle (p''_1 w''_1, \ldots, p''_n w''_n), l \rangle$ such that:
  $\langle (p_1 w_1, \ldots, p_n w_n), i \rangle \stackrel{k}{\Longrightarrow}_M \langle (p''_1 w''_1, \ldots, p''_n w''_n), l \rangle$; $\langle (p''_1 w''_1, \ldots, p''_n w''_n), l \rangle \Rightarrow_M \langle (p''_1 w''_1, \ldots, p''_n w''_n), j \rangle$; and $(p''_1 w''_1, \ldots, p''_n w''_n) \Longrightarrow_{N_j}^* (p'_1 w'_1, \ldots, p'_n w'_n)$.

We extend $\stackrel{k}{\Longrightarrow}_M$ to configurations as follows:
$(p_1 w_1, \ldots, p_n w_n) \stackrel{k}{\Longrightarrow}_M (p'_1 w'_1, \ldots, p'_n w'_n)$ iff there exist two phase indices i and j such that $\langle (p_1 w_1, \ldots, p_n w_n), i \rangle \stackrel{k}{\Longrightarrow}_M \langle (p'_1 w'_1, \ldots, p'_n w'_n), j \rangle$.

The k-bounded switch reachability problem for MAPNs between two sets of (indexed) configurations C and C' consists of determining whether there are $c \in C$ and $c' \in C'$ such that $c \overset{k}{\Longrightarrow}_M c'$. Intuitively, this means that $c \overset{k}{\Longrightarrow}_M c'$ iff the (indexed) configuration c' can be reached from c after switching the phase of the network at most k times according to the phase graph T. In this case, we say that c' is k-bounded reachable from c.

Unfortunately, even k-bounded switch reachability is undecidable for cyclic as well as acyclic MAPNs. Indeed, it is easy to see that performing k-bounded reachability in M amounts to performing "unrestricted" reachability in the acyclic network defined by $(N_1, \ldots, N_m, T_k)$, where T_k is obtained by considering all the possible paths of T having at most k transitions. Therefore, we have the following from Theorem 13.

Corollary 2 ([3, 69]) *The k-bounded reachability problem between recognizable sets of (indexed) configurations is undecidable for MAPNs. This holds even for $k = 1$.*

However, we have the following from Corollary 1 and the observation above.

Corollary 3 ([3, 69]) *The k-bounded switch reachability problem between recognizable sets of (indexed) configurations is decidable for* stable *MAPNs.*

18.4.2.3 A Semi-algorithm for k-Bounded Reachability for MAPNs

The result above can be used to construct a semi-decision procedure for the k-bounded switch reachability problem for *general* MAPNs. Let $M = (N_1, \ldots, N_m, T)$ be an MAPN. The idea consists of taking advantage of the fact that k-bounded switch reachability is decidable for *stable* networks. To do so, we compute a stable network $M' = (N'_1, \ldots, N'_{m'}, T')$ s.t. the processes in M' have the same behaviors as in M but can perform more phase switches. This ensures that given two configurations c and c', $c \overset{k}{\Longrightarrow}_{M'} c'$ implies that there exists k' such that $c \overset{k'}{\Longrightarrow}_M c'$. This gives the semi-decision procedure since we can decide k-bounded reachability for M' thanks to its stability.

To compute the stable network M', the idea consists of decomposing every acyclic PDN N_j $(j \leq m)$ into stable subnetworks $N_j^1, \ldots, N_j^{i_j}$ such that the behavior of each subnetwork N_j^l is also a behavior of N_j, and such that any behavior of N_j can be obtained by performing a number of switches between the different N_j^l's. The computed network satisfies the following.

Theorem 15 ([3, 69]) *Let C and C' be two recognizable sets of (indexed) configurations. Then if C' is k-bounded reachable from C by M, there exists $k' \geq k$ such that C' is k'-bounded reachable from C by M'.*

18.4.2.4 A Semi-algorithm for the Reachability Problem for General PDNs

The previous results on bounded phase switch reachability for MAPNs can be used to derive a semi-algorithm to check reachability for general PDNs (even cyclic ones). Let $N = (P_1, \ldots, P_n, R)$ be a PDN, where for i, $1 \leq i \leq n$, $P_i = (Q_i, \Gamma_i, \Delta_i)$ is a communicating pushdown system. The construction underlying Theorem 12 produces an MAPN M such that reachability in N can be reduced to reachability in M. Let C and C' be two recognizable sets of configurations of N. We can show that if C' is reachable from C in N, then there exists an index k such that C' is k-bounded reachable from C in M. Thus, the semi-algorithm given in the previous section can be used to check reachability in N, and thus in PDNs.

This technique generalizes the algorithms proposed in [7, 58] for bounded context-switch analysis. Indeed, the notion of phase is more general than the notions of context used in these works in the sense that, if we encode a PDN model in those proposed in [7, 58], one single phase may correspond to an unbounded number of context switches.

18.4.3 Threads Communicating via Lossy Channels

We now consider acyclic networks of pushdown systems communicating through unbounded lossy FIFO channels. An Acyclic Lossy Channel Pushdown Network (APN_{lc} for short) [2] is a tuple $H = (P_1, \ldots, P_n, C, M)$ where: (1) $C \subseteq \{(j, i) : 1 \leq i < j \leq n\}$ is a finite set of unidirectional channels,[1] (2) M is a finite set of messages, and (3) for every $i \in \{1, \ldots, n\}$, $P_i = (Q_i, \Sigma_i, \Gamma_i, \Delta_i)$ is a pushdown system, where Q_i is a finite set of states, $\Sigma_i = (\{!\} \times M \times \{j : (i, j) \in C\}) \cup (\{?\} \times M \times \{j : (j, i) \in C\}) \cup (\{nop\})$ is a finite set of transition labels, Γ_i is a finite stack alphabet, and Δ_i is a finite set of transition rules of the form: $\langle p, u\rangle \overset{a}{\hookrightarrow} \langle p', u'\rangle$ where $a \in \Sigma_i$, $p, p' \in Q_i$, and $u, u' \in \Gamma_i^*$ such that either (i) $|u| = 1$ and $u' = \epsilon$ (pop operation), (ii) $u = \epsilon$ and $|u'| = 1$ (push operation), or (3) $u = u' = \epsilon$ (no operation on the stack).

A transition of process P_i labeled by $(!, m, j)$ means "P_i sends message m via the channel $(i, j) \in C$ to process j", whereas a transition labeled by $(?, m, j)$ means "P_i receives message m from the channel $(j, i) \in C$ sent by j". A nop corresponds to an internal action.

A *configuration* of the APN_{lc} H is a vector $\langle p_1 w_1, \ldots, p_n w_n, \mathcal{V}\rangle$ where $p_i w_i \in Q_i \Gamma_i^*$ is a local configuration of process P_i and $\mathcal{V}$ is a mapping from C to M^* giving the contents of each channel, i.e., $\mathcal{V}(i, j)$ describes the content of the channel (i, j).

We define a *transition relation* $\Longrightarrow_H$ between configurations as follows: $\langle p_1 w_1, \ldots, p_n w_n, \mathcal{V}\rangle \Longrightarrow_H \langle p'_1 w'_1, \ldots, p'_n w'_n, \mathcal{V}'\rangle$ iff $\exists i \in \{1, \ldots, n\}$ and

[1] Notice that the graph defined by C is acyclic.

$\exists(\langle p,u\rangle \overset{a}{\hookrightarrow} \langle p',u'\rangle) \in \Delta_i$ such that (1) $p = p_i$ and $p' = p'_i$, (2) $w_i = uv$ and $w'_i = u'v$ for some $v \in \Gamma_i^*$, (3) $\forall j \neq i.\ p_j = p'_j$ and $w_j = w'_j$, and (4) either (i) $a = (?,m,k)$ is a *receive* operation; $m\mathscr{V}'(k,i) \preceq \mathscr{V}(k,i)$ and $\mathscr{V}'(j,l) \preceq \mathscr{V}(j,l)$ for every $(j,l) \in C$ such that $(j,l) \neq (k,i)$: message m is read from the channel (k,i), and the contents of all the channels can lose some messages since the channels are lossy (this is expressed by the subword relation $\preceq$), or (ii) $a = (!,m,k)$ is a *send* operation, and $\mathscr{V}'(k,i) \preceq \mathscr{V}(k,i)m$ and $\mathscr{V}'(j,l) \preceq \mathscr{V}(j,l)$ for every $(j,l) \in C$ such that $(j,l) \neq (k,i)$ (m is added to the channel (k,i) that receives the message. Moreover, all the channels can lose messages), or (iii) $a = nop$, and $\mathscr{V}'(j,l) \preceq \mathscr{V}(j,l)$ for every $(j,l) \in C$ (to express the loss of messages). Let $\Longrightarrow_H^*$ denote the reflexive transitive closure of $\Longrightarrow_H^*$. $post_H$ and $post_H^*$ are defined in the standard manner.

Theorem 16 ([2]) *The reachability problem between configurations* (*and therefore between recognizable sets*) *for* APN_{lc} *is decidable.*

18.5 Other Models: Finite State Systems and Sequential Programs

The previous sections introduced several PDS-based models and model-checking algorithms that have been proposed for verifying concurrent programs. As mentioned in the introduction, these algorithms provide results for exact model-checking in the limit, often focusing on decidability issues. There has been less work on applying them to concurrent programs in practice, partly due to the huge gap between high-level programming languages on one end and the low-level PDS-based models on the other.

In a broader context, concurrent programs are related to finite state concurrent systems, with the added twist that programs are recursive and possibly infinite state. As we saw earlier, the presence of communication/synchronization along with recursion often leads to undecidability. Therefore, a practical strategy is to ignore recursion altogether, relying on finite state models, or finite state abstractions of infinite state models. Typically, the setting either includes the assumption that the concurrent program is terminating, or applies some bounded analysis like testing or bounded model-checking. Therefore, unbounded recursion is not handled, and function/procedure bodies in the programs are usually inlined (up to some depth bound). On the other hand, concurrent programs are also related to sequential programs, with the added twist that the individual programs (threads or processes) interact with each other.

With the success of model-checking on finite state systems and sequential programs, it is no surprise that many efforts have tried to extend these to concurrent programs. In this section, we describe techniques inspired by these two domains. Again, our emphasis is on the main ideas that have been used to handle synchronization/communication and the complexity due to process or thread interleavings.

Our intent is not to provide a survey (also due to space limitations), but to provide pointers to representative efforts, since these ideas can be combined in various ways to develop practical verifiers for concurrent programs. Not all techniques deal precisely with recursion or with thread interactions, frequently using sound abstractions but giving up on completeness (even in decidable cases). Furthermore, most techniques described here do not address unbounded heap structures, which can be handled by combining them with shape analysis or techniques based on separation logic, e.g. [13, 31].

We will start by describing techniques that have worked well on concurrent finite state systems/abstractions. These include partial-order reduction (to reduce the number of interleavings, often in explicit search-based techniques) and use of symbolic memory constraints (to explore all interleavings, often using SMT solvers).

We will then describe techniques that handle recursive procedures, inspired by sequential program verification techniques such as abstract interpretation, dataflow analysis, and CEGAR-based model-checking. Here, effective reasoning about thread interference plays a key role. We discuss sequentialization, where control interference is converted to data nondeterminism and additional constraints are introduced to capture a given schedule. After that, we discuss automatic generation of invariants, which are used to refine over-approximations of thread interferences or to incrementally add them to under-approximations. The ultimate abstraction is where interference from other threads is regarded as the environment, in the setting of thread-modular or assume-guarantee compositional techniques for concurrent programs.

Finally, we discuss trace-based dynamic methods. Due to the scalability limitations of static verification, there has been increased interest in dynamic techniques for systematically exploring (parts of) concurrent programs. We discuss dynamic partial-order reduction and preemptive context-bounding techniques that compel the scheduler to dynamically explore a reduced set of interleavings. We also describe predictive analysis techniques, where a trace-based model derived from dynamic executions is used to predict concurrency bugs in alternate interleavings. Typically, these methods work for a given test input only, i.e. input coverage is limited, as in software testing.

18.5.1 Partial-Order Reduction

Model-checking based on partial-order reduction (POR) [29, 57, 71] exploits the observation that many interleavings in concurrent systems are equivalent. In particular, according to Mazurkiewicz's trace theory [53], two sequences are equivalent if they can be obtained from each other by successively permuting adjacent independent transitions. In the context of verifying multi-threaded programs, all operations are separated into *visible* and *non-visible* operations, based on whether they affect the state of variables that are shared between threads. Essentially, interleavings are considered only at the *transaction* boundaries, where a transaction captures a sequence

of thread-local operations. Two visible operations are considered dependent if they access the same shared variable, and if at least one of them is a write-operation.

However, computing a precise dependence relation may be as hard as verification itself. Therefore, POR-based methods often use a conservative static analysis to determine reachability of dependent transitions in the program. This was used successfully in several pioneering works on software model-checking [30, 37, 72], including verification of concurrent programs.

This reasoning can be made more precise by utilizing the techniques described in earlier sections on the control state space of the program, including semantics of synchronizing operations available in the associated programming language. For example, lock-based reasoning for Java programs [68] and lock acquisition histories [42] have been used to further reduce the number of interleavings, on top of static partial-order reduction.

To improve upon the imprecision due to static analysis, dynamic partial-order reduction [26] and Cartesian partial-order reduction [32] detect dependent transitions dynamically during model-checking, thereby leading to a greater reduction. In addition to these explicit state traversal techniques, partial-order reduction has been employed in symbolic settings to achieve optimal reduction [45, 78].

18.5.2 Symbolic Reasoning with Memory Consistency Constraints

Another interesting line of symbolic verification relies on using memory consistency axioms for deriving the feasible behaviors of a multi-threaded program. These axioms specify rules on when a shared memory write may be observed by a memory read. The transition relations of the threads, together with these axioms, are symbolically encoded as a logic formula. Then, SAT or SMT solvers are used to find violations (also symbolically encoded) among all interleavings that satisfy the memory consistency constraints.

This approach was used for symbolically detecting data races for the Java memory model [80], and for checking a given set of (bounded) tests on programs for weak memory models [10, 11]. It has also been used for bounded model-checking multi-threaded programs—with an *a priori* fixed bound on the number of context switches [59], or without an *a priori* fixed bound by using a token-based encoding to enforce sequential consistency [27, 28]. The basic approach has also been combined with thread-modular summarization [65]. To improve scalability, the notion of *interference abstraction* and refinement has been introduced to weaken/strengthen the memory consistency axioms [66].

There are some differences among these efforts in the details of the symbolic encodings and additional optimizations, where the number of memory consistency constraints ranges from quadratic to cubic in the number of shared variable accesses. Overall, they provide symbolic frameworks to uniformly combat the state space explosion due to (shared) data as well as interleavings, and effectively exploit the advances in SAT and SMT solvers.

18.5.3 Concurrent Dataflow Analysis and Invariant Generation

Inspired by the success of dataflow analysis and abstract interpretation techniques on sequential programs, there have been several efforts to extend them to concurrent programs, bringing in their inherent machinery such as abstractions to handle the large/infinite data spaces and summarization techniques for handling procedures.

A convenient *sequentialization* technique effectively allows any sequential dataflow analysis technique to be utilized on concurrent programs, under the assumption of bounded context switching. This was motivated by a useful decidability result [58], which showed that context-bounded analysis of arbitrary concurrent programs is decidable. The main idea in sequentialization is to introduce nondeterministic data variables to capture the effect of thread (control) interference, followed by enforcing constraints on these variable values according to some bounded schedule [50]. This is similar in spirit to the symbolic encoding of memory consistency constraints described in the previous subsection. However, the ability to plug in any sequential program analysis supports a rich set of features and techniques optimized over the years. An example application was shown on systems code [48].

Another technique that also focuses on thread interference is based on using a *transaction graph* that over-approximates the set of interleavings between given threads, and iteratively refines it by employing invariants automatically generated by performing abstract interpretation on this graph [43]. The abstract interpretation can use various domains with increasing precision (intervals, octagons, polyhedra). To help generate more precise invariants, the technique also uses partial-order reduction and synchronization constraints, specifically PDS-based reasoning on locks, to create the initial transaction graph.

Indeed, many static analysis approaches that ignore the effects of synchronization operations can improve their precision by utilizing the PDS-based techniques described in Sect. 18.3. For example, reasoning about nested locks has also been utilized for compositional bitvector dataflow analysis for concurrent programs [24].

Instead of starting from a large set of interleavings and then refining to reduce them, a different approach can be taken to propagate the interferences incrementally [54]. This approach also has the benefit of utilizing standard abstract domains (e.g. intervals, octagons) for computing the fixpoints for individual threads. The scheduler states are partitioned to handle the interference effects based on mutual exclusion and scheduling priorities. After propagating the interference effects, the individual thread fixpoint computations are repeated, and the entire process is iterated until thread interferences stabilize.

The ultimate abstraction is to view interference from other threads as the environment. Compositional rely-guarantee reasoning can then be used to find the environment invariants needed to establish the correctness property, with the advantage that thread-modular verification is more scalable, in general, than handling a monolithic global system. The automatic discovery of environment invariants for compositional verification can be performed by various techniques, e.g., use of counterexample-guided abstraction refinement [20, 34, 36], transition predicate abstraction refinement [33], learning-based methods to automatically generate com-

ponent interfaces [35, 63], etc. The relevant issues to consider are whether the discovered invariants are modular (i.e., refer to only shared variables) or non-modular (i.e., refer to local states of other threads also) and whether the discovery method is complete (i.e., it will discover the invariants if they exist).

18.5.4 Trace-Based Dynamic Model-Checking

Finally, we briefly describe trace-based methods for finding bugs in concurrent programs. Due to the fact that these methods work with traces (or models derived from traces), they can only reason about parts of the program state space at a time. However, for large programs, this often becomes an enabler to overcome the scalability issue. The traces are typically derived from actual program executions on given test cases, and there is a rich history of dynamic analysis techniques to find concurrency bugs (outside the scope of this chapter). Here, we describe the application of static verification techniques on trace-based models, where the main motivation is to explore alternate interleavings (thread schedules) of the events observed during the test run.

Dynamic partial-order reduction (DPOR) [26] uses stateless model-checking on given test cases to explore all non-redundant interleavings. This has also been customized in a multi-process setting to verify MPI programs [73]. However, the set of all non-redundant interleavings can be quite large in practice. This motivated the development of preemptive context bounding (PCB) [55], which explores interleavings with some fixed bound on the number of context switches. The intuition here is that many bugs can be detected within a small bound, which seems to hold in practice. Other heuristics for schedule selection can be added on top of DPOR [77] to improve performance.

Both DPOR and PCB work by taking control of the scheduler, and execute the actual program to explore alternate interleavings. Alternately, it is possible to derive models from the executed trace, for the purpose of static exploration. These models are called *predictive models*, in that they are used to *predict* property violations in alternate interleavings of events in the given trace. Different models and checkers have been proposed for detecting data races, atomicity and serializability violations, and nondeterminism. Depending on the precision of the predictive models and the related analysis, the methods provide different coverage over the space of alternate interleavings, and may report only true violations or potentially false violations also.

Typically, predictive analysis methods based on Lamport's happens-before causality relation and its extensions [17, 52, 62, 81] report only true violations, but their predictive coverage may be small due to enforcing the observed causalities. In particular, most of these techniques model lock/unlock operations as happens-before according to the order in which locks are acquired in the observed trace. They do not consider alternate orders on acquiring locks.

In contrast, other predictive methods based on synchronization constraints, e.g., based on lock acquisition histories [25] or universal causality graphs [44], provide

higher predictive coverage but may report false violations (unless followed by additional feasibility checks). The reason is that these methods typically do not track dataflow constraints. Some efforts have also combined happens-before reasoning with precise reasoning on locks [64]. Note that locks enforce mutual exclusion, but do not enforce any particular order on acquiring locks. To allow exploration of alternate orders, some amount of search is needed in general.[2]

The extreme along both axes—full predictive coverage and highest precision—is obtained by tracking control flow, dataflow, and synchronization constraints [75, 76]. The proposed predictive model, called a concurrent trace program (CTP), encodes inter-thread data flow, concurrency primitives, and correctness properties uniformly as symbolic happens-before constraints. An SMT solver is used to find violations, performing a symbolic search over thread interleavings that are guaranteed to be executable.

18.5.5 Other Related Techniques

Due to reasons of brevity, this section focused on some essential ideas in concurrent program verification arising from finite state or program verification. There are several topics of related research that we are unable to cover: verification of parallel and message-passing programs for high-performance computing [5, 73, 79]; automatic synthesis of synchronization operations [14, 18, 22, 47, 67]; verification of memory models (sequential consistency, weaker and relaxed memory models) [1, 12]; correctness proofs of concurrent data structures [21, 70]; use of separation logic for concurrent programs [56]. The interested reader can follow some representative pointers provided above.

References

1. Alglave, J., Maranget, L., Sarkar, S., Sewell, P.: Fences in weak memory models (extended version). Form. Methods Syst. Des. **40**(2), 170–205 (2012)
2. Atig, M.F., Bouajjani, A., Touili, T.: On the reachability analysis of acyclic networks of pushdown systems. In: van Breugel, F., Chechik, M. (eds.) CONCUR. LNCS, vol. 5201, pp. 356–371 (2008)
3. Atig, M.F., Touili, T.: Verifying parallel programs with dynamic communication structures. In: Maneth, S. (ed.) CIAA. LNCS, vol. 5642, pp. 145–154. Springer, Heidelberg (2009)
4. Bonnet, R., Chadha, R., Madhusudan, P., Viswanathan, M.: Reachability under contextual locking. Log. Methods Comput. Sci. **9**(3), 1–17 (2013)
5. Bouajjani, A., Emmi, M.: Bounded phase analysis of message-passing programs. Int. J. Softw. Tools Technol. Transf. **16**(2), 127–146 (2014)

[2] Acyclicity analysis on Universal Causality Graphs [44] is sound and complete for reachability with two threads, but it was shown to be not complete for more than two threads [64].

6. Bouajjani, A., Esparza, J., Maler, O.: Reachability analysis of pushdown automata: application to model-checking. In: Marzurkiewicz, A., Winkowski, J. (eds.) CONCUR. LNCS, vol. 1243, pp. 135–150. Springer, Heidelberg (1997)
7. Bouajjani, A., Esparza, J., Schwoon, S., Strejcek, J.: Reachability analysis of multithreaded software with asynchronous communication. In: Ramanujan, R., Sen, S. (eds.) FSTTCS. LNCS, vol. 3821, pp. 348–359. Springer, Heidelberg (2005)
8. Bouajjani, A., Esparza, J., Touili, T.: A generic approach to the static analysis of concurrent programs with procedures. In: POPL, pp. 62–73. ACM, New York (2003)
9. Bouajjani, A., Esparza, J., Touili, T.: A generic approach to the static analysis of concurrent programs with procedures. Int. J. Found. Comput. Sci. **14**(4), 551–582 (2003)
10. Burckhardt, S., Alur, R., Martin, M.M.K.: Bounded model checking of concurrent data types on relaxed memory models: a case study. In: Ball, T., Jones, R.B. (eds.) CAV. LNCS, vol. 4144, pp. 489–502. Springer, Heidelberg (2006)
11. Burckhardt, S., Alur, R., Martin, M.M.K.: Checkfence: checking consistency of concurrent data types on relaxed memory models. In: PLDI, pp. 12–21. ACM, New York (2007)
12. Burckhardt, S., Musuvathi, M.: Effective program verification for relaxed memory models. In: Gupta, A., Malik, S. (eds.) CAV. LNCS, vol. 5123, pp. 107–120. Springer, Heidelberg (2008)
13. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. J. ACM **58**(6), 26 (2011)
14. Cerný, P., Henzinger, T.A., Radhakrishna, A., Ryzhyk, L., Tarrach, T.: Efficient synthesis for concurrency by semantics-preserving transformations. In: Sharygina, N., Veith, H. (eds.) CAV. LNCS, vol. 8044, pp. 951–967. Springer, Heidelberg (2013)
15. Chadha, R., Madhusudan, P., Viswanathan, M.: Reachability under contextual locking. In: Flanagan, C., König, B. (eds.) TACAS. LNCS, vol. 7214, pp. 437–450. Springer, Heidelberg (2012)
16. Chaki, S., Clarke, E.M., Kidd, N., Reps, T.W., Touili, T.: Verifying concurrent message-passing C programs with recursive calls. In: Hermanns, H., Palsberg, J. (eds.) TACAS. LNCS, vol. 3920, pp. 334–349. Springer, Heidelberg (2006)
17. Chen, F., Rosu, G.: Parametric and sliced causality. In: Damm, W., Hermanns, H. (eds.) CAV. LNCS, vol. 4590, pp. 240–253. Springer, Heidelberg (2007)
18. Cherem, S., Chilimbi, T.M., Gulwani, S.: Inferring locks for atomic sections. In: PLDI, pp. 304–315. ACM, New York (2008)
19. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching-time temporal logic. In: Kozen, D. (ed.) Logics of Programs. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1981)
20. Cohen, A., Namjoshi, K.S.: Local proofs for global safety properties. Form. Methods Syst. Des. **34**(2), 104–125 (2009)
21. Elmas, T., Qadeer, S., Tasiran, S.: A calculus of atomic actions. In: POPL, pp. 2–15. ACM, New York (2009)
22. Emmi, M., Fischer, J.S., Jhala, R., Majumdar, R.: Lock allocation. In: POPL, pp. 291–296. ACM, New York (2007)
23. Esparza, J., Schwoon, S.: A BDD-based model checker for recursive programs. In: Berry, G., Common, N., Finkel, A. (eds.) CAV. LNCS, vol. 2102, pp. 324–336. Springer, Heidelberg (2001)
24. Farzan, A., Kincaid, Z.: Compositional bitvector analysis for concurrent programs with nested locks. In: Cousot, R., Martel, M. (eds.) SAS. LNCS, vol. 6337, pp. 253–270. Springer, Heidelberg (2010)
25. Farzan, A., Madhusudan, P., Sorrentino, F.: Meta-analysis for atomicity violations under nested locking. In: Bonajjani, A., Maler, O. (eds.) CAV. LNCS, vol. 5643, pp. 248–262. Springer, Heidelberg (2009)
26. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: POPL, pp. 110–121. ACM, New York (2005)

27. Ganai, M.K., Gupta, A.: Efficient modeling of concurrent systems in BMC. In: Havelund, K., Majumdar, R., Palsberg, J. (eds.) SPIN. LNCS, vol. 5156, pp. 114–133. Springer, Heidelberg (2008)
28. Ganai, M.K., Kundu, S.: Reduction of verification conditions for concurrent system using mutually atomic transactions. In: Păsăreanu, C. (ed.) SPIN. LNCS, vol. 5578, pp. 68–87. Springer, Heidelberg (2009)
29. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems—An Approach to the State-Explosion Problem. LNCS, vol. 1032. Springer, Heidelberg (1996)
30. Godefroid, P.: Model checking for programming languages using Verisoft. In: POPL, pp. 174–186. ACM, New York (1997)
31. Gotsman, A., Berdine, J., Cook, B., Sagiv, M.: Thread-modular shape analysis. In: PLDI, pp. 266–277. ACM, New York (2007)
32. Gueta, G., Flanagan, C., Yahav, E., Sagiv, M.: Cartesian partial-order reduction. In: Bošnački, D., Edelkamp, S. (eds.) SPIN Workshop on Model Checking Software. LNCS, vol. 4595, pp. 95–112. Springer, Heidelberg (2007)
33. Gupta, A., Popeea, C., Rybalchenko, A.: Predicate abstraction and refinement for verifying multi-threaded programs. In: POPL, pp. 331–344. ACM, New York (2011)
34. Henzinger, T.A., Jhala, R., Majumdar, R.: Race checking by context inference. In: PLDI, pp. 1–13. ACM, New York (2004)
35. Henzinger, T.A., Jhala, R., Majumdar, R.: Permissive interfaces. In: ESEC/SIGSOFT FSE, pp. 31–40. ACM, New York (2005)
36. Henzinger, T.A., Jhala, R., Majumdar, R., Qadeer, S.: Thread-modular abstraction refinement. In: Hunt, W.A. Jr., Somenzi, F. (eds.) CAV. LNCS, vol. 2725, pp. 262–274. Springer, Heidelberg (2003)
37. Holzmann, G.J.: Software model checking with SPIN. Adv. Comput. **65**, 78–109 (2005)
38. Kahlon, V.: Boundedness vs. unboundedness of lock chains: characterizing decidability of pairwise CFL-reachability for threads communicating via locks. In: LICS, pp. 27–36. IEEE, Piscataway (2009)
39. Kahlon, V.: Reasoning about threads with bounded lock chains. In: Katoen, J., König, B. (eds.) CONCUR. LNCS, vol. 6901, pp. 450–465. Springer, Heidelberg (2011)
40. Kahlon, V., Gupta, A.: An automata-theoretic approach for model checking threads for LTL properties. In: LICS, pp. 101–110. IEEE, Piscataway (2006)
41. Kahlon, V., Gupta, A.: On the analysis of interacting pushdown systems. In: POPL, pp. 303–314. ACM, New York (2007)
42. Kahlon, V., Ivančić, F., Gupta, A.: Reasoning about threads communicating via locks. In: Etessami, K., Rajamani, S.K. (eds.) CAV. LNCS, vol. 3576, pp. 505–518. Springer, Heidelberg (2005)
43. Kahlon, V., Sankaranarayanan, S., Gupta, A.: Semantic reduction of thread interleavings in concurrent programs. In: Kowalewski, S., Philippou, A. (eds.) TACAS. LNCS, vol. 5505, pp. 124–138. Springer, Heidelberg (2009)
44. Kahlon, V., Wang, C.: Universal causality graphs: a precise happens-before model for detecting bugs in concurrent programs. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV. LNCS, vol. 6174, pp. 434–449. Springer, Heidelberg (2010)
45. Kahlon, V., Wang, C., Gupta, A.: Monotonic partial order reduction: an optimal symbolic partial order reduction technique. In: Bouajjani, A., Maler, O. (eds.) CAV. LNCS, vol. 5643, pp. 398–413. Springer, Heidelberg (2009)
46. Kidd, N., Lammich, P., Touili, T., Reps, T.W.: A decision procedure for detecting atomicity violations for communicating processes with locks. Int. J. Softw. Tools Technol. Transf. **13**(1), 37–60 (2011)
47. Kuperstein, M., Vechev, M.T., Yahav, E.: Automatic inference of memory fences. SIGACT News **43**(2), 108–123 (2012)
48. Lahiri, S.K., Qadeer, S., Rakamaric, Z.: Static and precise detection of concurrency errors in systems code using SMT solvers. In: Bouajjani, A., Maler, O. (eds.) CAV. LNCS, vol. 5643, pp. 509–524. Springer, Heidelberg (2009)

49. Lal, A., Balakrishnan, G., Reps, T.: Extended weighted pushdown systems. In: Ekessami, K., Rajamani, S.K. (eds.) CAV. LNCS, vol. 3576, pp. 434–448. Springer, Heidelberg (2005)
50. Lal, A., Reps, T.W.: Reducing concurrent analysis under a context bound to sequential analysis. In: Gupta, A., Malik, S. (eds.) CAV. LNCS, vol. 5123, pp. 37–51. Springer, Heidelberg (2008)
51. Lammich, P., Müller-Olm, M., Wenner, A.: Predecessor sets of dynamic pushdown networks with tree-regular constraints. In: Bouajjani, A., Maler, O. (eds.) CAV. LNCS, vol. 5643, pp. 525–539. Springer, Heidelberg (2009)
52. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (1978)
53. Mazurkiewicz, A.W.: Trace theory. In: Brauer, W., Reisig, W., Rozenberg, G. (eds.) Advances in Petri Nets. LNCS, vol. 255, pp. 279–324. Springer, Heidelberg (1986)
54. Miné, A.: Static analysis of run-time errors in embedded critical parallel C programs. In: Barté, G. (ed.) ESOP. LNCS, vol. 6602. Springer, Heidelberg (2011)
55. Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P.A., Neamtiu, I.: Finding and reproducing Heisenbugs in concurrent programs. In: OSDI, pp. 267–280. USENIX Association, Berkeley (2008)
56. O'Hearn, P.W.: Resources, concurrency and local reasoning. In: Gardner, P., Yoshida, N. (eds.) CONCUR. LNCS, vol. 3170, pp. 49–67. Springer, Heidelberg (2004)
57. Peled, D.: All from one, one for all: on model checking using representatives. In: Courcoubetis, C. (ed.) CAV. LNCS, vol. 697, pp. 409–423. Springer, Heidelberg (1993)
58. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: Halbwachs, N., Zuck, L. (eds.) TACAS. LNCS, vol. 3440, pp. 93–107. Springer, Heidelberg (2005)
59. Rabinovitz, I., Grumberg, O.: Bounded model checking of concurrent programs. In: Etessami, K., Rajamani, S.K. (eds.) CAV. LNCS, vol. 3576, pp. 82–97. Springer, Heidelberg (2005)
60. Ramalingam, G.: Context-sensitive synchronization-sensitive analysis is undecidable. In: TOPLAS, pp. 416–430. ACM, New York (2000)
61. Reps, T., Schwoon, S., Jha, S.: Weighted pushdown systems and their application to interprocedural dataflow analysis. In: Cousot, R. (ed.) SAS. LNCS, vol. 2694, pp. 189–213. Springer, Heidelberg (2003)
62. Sen, K., Rosu, G., Agha, G.: Runtime safety analysis of multithreaded programs. In: ESEC/SIGSOFT FSE, pp. 337–346. ACM, New York (2003)
63. Singh, R., Giannakopoulou, D., Pasareanu, C.S.: Learning component interfaces with may and must abstractions. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV. LNCS, vol. 6174, pp. 527–542. Springer, Heidelberg (2010)
64. Sinha, A., Malik, S., Gupta, A.: Efficient predictive analysis for detecting nondeterminism in multi-threaded programs. In: FMCAD, pp. 6–15. IEEE, Piscataway (2012)
65. Sinha, N., Wang, C.: Staged concurrent program analysis. In: SIGSOFT FSE, pp. 47–56. ACM, New York (2010)
66. Sinha, N., Wang, C.: On interference abstractions. In: POPL, pp. 423–434. ACM, New York (2011)
67. Solar-Lezama, A., Jones, C.G., Bodík, R.: Sketching concurrent data structures. In: PLDI, pp. 136–148. ACM, New York (2008)
68. Stoller, S.D.: Model-checking multi-threaded distributed Java programs. Int. J. Softw. Tools Technol. Transf. **4**(1), 71–91 (2002)
69. Touili, T., Atig, M.F.: Verifying parallel programs with dynamic communication structures. Theor. Comput. Sci. **411**(38–39), 3460–3468 (2010)
70. Vafeiadis, V.: Automatically proving linearizability. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV. LNCS, vol. 6174, pp. 450–464. Springer, Heidelberg (2010)
71. Valmari, A.: Stubborn sets for reduced state space generation. In: Rozenberg, G. (ed.) Advances in Petri Nets. LNCS, vol. 483, pp. 491–515. Springer, Heidelberg (1989)
72. Visser, W., Havelund, K., Brat, G.P., Park, S., Lerda, F.: Model checking programs. Autom. Softw. Eng. **10**(2), 203–232 (2003)

73. Vo, A., Vakkalanka, S.S., Delisi, M., Gopalakrishnan, G., Kirby, R.M., Thakur, R.: Formal verification of practical MPI programs. In: PPOPP, pp. 261–270. ACM, New York (2009)
74. Walukiewicz, I.: Model checking CTL properties of pushdown systems. In: Kapoor, S., Prasad, S. (eds.) FSTTCS. LNCS, vol. 1974, pp. 127–138. Springer, Heidelberg (2000)
75. Wang, C., Kundu, S., Ganai, M.K., Gupta, A.: Symbolic predictive analysis for concurrent programs. In: Cavalcanti, A., Dams, D. (eds.) FM. LNCS, vol. 5850, pp. 256–272. Springer, Heidelberg (2009)
76. Wang, C., Limaye, R., Ganai, M.K., Gupta, A.: Trace-based symbolic analysis for atomicity violations. In: Esparza, J., Majumdar, R. (eds.) TACAS. LNCS, vol. 6015, pp. 328–342. Springer, Heidelberg (2010)
77. Wang, C., Said, M., Gupta, A.: Coverage guided systematic concurrency testing. In: ICSE, pp. 221–230. ACM, New York (2011)
78. Wang, C., Yang, Z., Kahlon, V., Gupta, A.: Peephole partial order reduction. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS. LNCS, vol. 4963, pp. 382–396. Springer, Heidelberg (2008)
79. Wies, T., Zufferey, D., Henzinger, T.A.: Forward analysis of depth-bounded processes. In: Ong, L. (ed.) FOSSACS. LNCS, vol. 6014, pp. 94–108. Springer, Heidelberg (2010)
80. Yang, Y., Gopalakrishnan, G., Lindstrom, G.: Memory-model-sensitive data race analysis. In: Davies, J., Schutte, W., Barnett, M. (eds.) ICFEM. LNCS, vol. 3308, pp. 30–45 (2004)
81. Yi, J., Sadowski, C., Flanagan, C.: SideTrack: generalizing dynamic atomicity analysis. In: PADTAD, pp. 8:1–8:10. ACM, New York (2009)

Chapter 19
Combining Model Checking and Testing

Patrice Godefroid and Koushik Sen

Abstract Model checking and testing have a lot in common. Over the last two decades, significant progress has been made on how to broaden the scope of model checking from finite-state abstractions to actual software implementations. One way to do this consists of adapting model checking into a form of systematic testing that is applicable to industrial-size software. This chapter presents an overview of this strand of software model checking.

19.1 Introduction

Model checking and testing are similar in many ways. In practice, the main value of both is *to find bugs* in programs. And, if no bugs are to be found, both techniques increase the confidence that the program is correct.

In theory, model checking is a form of formal verification based on exhaustive state-space exploration. As famously stated by Dijkstra decades ago, *"testing can only find bugs, not prove their absence"*. In contrast, verification (including exhaustive testing) can prove the absence of bugs. This is the key feature that distinguishes verification, including model checking, from testing.

In practice, however, the verification guarantees provided by model checking are often limited: model checking checks only a program, or a manually written model of a program, for some specific properties, under some specific environmental assumptions, and the checking itself is usually approximate for nontrivial programs and properties when an exact answer is too expensive to compute. Therefore, model checking should be viewed in practice more as a form of *"super testing"* rather than as a form of formal verification in the strict mathematical sense. Compared to testing, model checking provides better coverage, but is more computationally expensive. Compared to more general forms of program verification like interactive

P. Godefroid (✉)
Microsoft Research, Redmond, WA, USA
e-mail: pg@microsoft.com

K. Sen
University of California, Berkeley, Berkeley, CA, USA

E.M. Clarke et al. (eds.), *Handbook of Model Checking*,
DOI 10.1007/978-3-319-10575-8_19

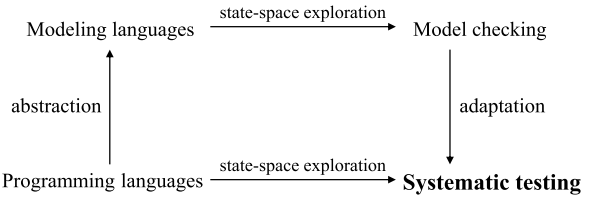


Fig. 1 Two main approaches to software model checking

theorem proving, model checking provides more limited verification guarantees, but is cheaper due to its higher level of automation. Model checking thus offers an *attractive practical trade-off* between testing and formal verification.

The key practical strength of model checking is that it is able to *find bugs* that would be extremely hard to find (and reproduce) with traditional testing. This key strength has been consistently demonstrated, over and over again, during the last three decades when applying model-checking tools to check the correctness of hardware and software designs, and more recently software implementations. It also explains the gradual adoption of model checking in various industrial environments over the last 20 years (hardware industry, safety-critical systems, software industry).

What prevents an even wider adoption of model checking is its relative *higher cost* compared to basic testing. This is why model checking has been adopted so far mostly in *niche* yet *critical* application domains where the cost of bugs is high enough to justify the cost of using model checking (hardware designs, communication switches, embedded systems, operating-system device drivers, security, etc.).

Over the last two decades, significant progress has been made on how to lower the cost of adoption of model checking even further when applied to software through the advent of *software model checking*. Unlike traditional model checking, a software model checker does not require a user to manually write an abstract *model* of the software program to be checked in some modeling language, but instead works directly on a program *implementation* written in a full-fledged programming language.

As illustrated in Fig. 1, there are essentially *two main approaches to software model checking*, i.e., two ways to broaden the scope of model checking from modeling languages to programming languages. One approach uses *abstraction*: it consists of automatically extracting an abstract model out of a software application by statically analyzing its code, and then analyzing this model using traditional model-checking algorithms (e.g., [4, 45, 85, 111]). Another approach uses *adaptation*: it consists of adapting model checking into a form of systematic testing that is applicable to industrial-size software (e.g., [60, 68, 108, 148]).

The aim of this chapter is to present an overview of this second approach to software model checking. We describe the main ideas and techniques used to systematically test and explore the state spaces of concurrent (Sect. 19.2) or data-driven (Sect. 19.3) software, or both (Sect. 19.4). We also discuss other related work (Sect. 19.5), such as combining systematic testing with static program analysis, runtime verification, and other testing techniques. However, this chapter is only meant to provide an introduction to this research area, it is *not* an exhaustive survey.

19.2 Systematic Testing of Concurrent Software

In this section, we present techniques inspired by model checking for systematically testing concurrent software. We discuss nondeterminism due to concurrency before nondeterminism due to data inputs (in the next section) for historic reasons. Indeed, model checking was first conceived for reasoning about concurrent reactive systems [38, 124], and software model checking via systematic testing was also first proposed for concurrent programs [60].

19.2.1 Classical Model Checking

Traditional model checking checks properties of a system modeled in some modeling language, typically some kind of notation for communicating finite-state machines. Given such a system's model, the formal semantics of the modeling language defines the *state space* of the model typically as some kind of *product* of the communicating finite-state machines modeling the system's components. A state space is usually defined as a directed graph whose nodes are states of the entire system and edges represent state changes. Branching in the graph represents either branching in individual state machine components or nondeterminism due to concurrency, i.e., different orderings of actions performed by different components. The state space of a system's model thus represents the joint dynamic behavior of all components interacting with each other in all possible ways. By systematically exploring its state space, model checking can reveal unexpected possible interactions between components of the system's model, and hence reveal potential flaws in the actual system.

Many properties of a system's model can be checked by exploring its state space: one can detect deadlocks, dead code, violations of user-specified assertions, etc. Moreover, the range of properties that state-space exploration techniques can verify has been substantially broadened during the last three decades thanks to the development of model-checking methods for various temporal logics (e.g., [39, 99, 124, 145]). Historically, the term "model checking" was introduced to mean "check whether a system is a model of a temporal logic formula," in the classic logical sense. This definition does not imply that a "model," i.e., an abstraction, of a system is checked. In this chapter, we will use the term "model checking" in a broad sense, to denote any systematic state-space exploration technique that can be used for verification purposes when it is exhaustive.

19.2.2 Software Model Checking Using a Dynamic Semantics

Just as a traditional model checker explores the state space of a system modeled as the product of concurrent finite-state components, one can systematically explore the *"product"* of concurrently executing *operating-system processes* by using a *run-*

time scheduler for driving the entire software application through the states and transitions of its state space [60].

The product, or *state space*, of concurrently executing processes can be defined *dynamically* as follows. Consider a concurrent system composed of a finite set of *processes* and a finite set of *communication objects*. Each process executes a sequence of *operations* described in a sequential program written in any full-fledged programming language (such as C, C++, etc.). Such sequential programs are *deterministic*: every execution of the program on the same input data performs the same sequence of operations. We assume that processes communicate with each other by performing *atomic operations* on communication objects. Examples of communication objects are shared variables, semaphores, and FIFO buffers. Operations on communication objects are called *visible operations*, while other operations are by default called *invisible*. The execution of an operation is said to be *blocking* if it cannot be completed; for instance, waiting for the reception of a message blocks until a message is received. We assume that only executions of visible operations may be blocking.

At any time, the concurrent system is said to be in a *state*. The system is said to be in a *global state* when the next operation to be executed by every process in the system is a visible operation. Every process in the system is always expected to eventually attempt to execute a visible operation.[1] This assumption implies that initially, after the creation of all the processes of the system, the system can reach a first and unique global state s_0, called the *initial global state* of the system.

A *process transition*, or *transition* for short, is defined as one visible operation followed by a *finite* sequence of invisible operations performed by a *single* process and ending just before a visible operation. Let T denote the set of all transitions of the system.

A transition is said to be *disabled* in a global state s when the execution of its visible operation is blocking in s. Otherwise, the transition is said to be *enabled* in s. A transition t enabled in a global state s can be *executed* from s. Since the number of invisible operations in a transition is finite, the execution of an enabled transition always terminates. When the execution of t from s is completed, the system reaches a global state s', called the *successor* of s by t and denoted by $s \xrightarrow{t} s'$.[2]

We can now define the *state space* of a concurrent system satisfying our assumptions as the transition system $A_G = (S, \Delta, s_0)$ representing its set of reachable global states and the (process) transitions that are possible between these:

- S is the set of global states of the system,
- $\Delta \subseteq S \times S$ is the *transition relation* defined as follows:

$$(s, s') \in \Delta \quad \text{iff} \quad \exists t \in T : s \xrightarrow{t} s',$$

- s_0 is the initial global state of the system.

[1] If a process does not attempt to execute a visible operation within a given amount of time, an error is reported at run-time.

[2] Operations on objects (and hence transitions) are deterministic: the execution of a transition t in a state s leads to a *unique* successor state.

We emphasize that an element of Δ, or *state-space transition*, corresponds to the execution of a single process transition $t \in T$ of the system. Remember that here we use the term "transition" to refer to a process transition, not to a state-space transition. Note how (process) transitions are defined as maximal sequences of interprocess "local" operations from one visible operation to the next. Interleavings of those local operations are not considered as part of the state space.

It can be proved [60] that, for any concurrent system satisfying the above assumptions, exploring only all its global states is sufficient to detect all its *deadlocks* and *assertion violations*, i.e., exploring all its non-global states is not necessary. This result justifies the choice of the specific dynamic semantics described in this section. Deadlocks are states where the execution of the next operation of every process in the system is blocking. Deadlocks are a notorious problem in concurrent systems, and can be difficult to detect through conventional testing. Assertions can be specified by the user in the code of any process with the special visible operation "assert". It takes as its argument a boolean expression that can test and compare the value of variables and data structures *local* to the process. Many undesirable system properties, such as unexpected message receptions, buffer overflows and application-specific error conditions, can easily be expressed as assertion violations.

Note that we consider here *closed* concurrent systems, where the environment of each process is formed by the other processes in the system. This implies that, in the case of a single "open" reactive system, the environment in which this system operates has to be represented somehow, possibly using other processes. In practice, a complete representation of such an environment may not be available, or may be very complex. It is then convenient to use a simplified representation of the environment, a *test driver* or *mock-up*, to simulate its behavior. For this purpose, it is useful to introduce a special operation to express a valuable feature of modeling languages not found in programming languages: *nondeterminism*. This operation, let us call it `nondet`,[3] takes as argument a positive integer n, and returns an integer in $[0, n]$. The operation is visible and nondeterministic: the execution of a transition starting with `nondet(`n`)` may yield up to $n+1$ different successor states, corresponding to different values returned by `nondet`. This operation can be used to represent *input data nondeterminism* or the effect of input data on the control flow of a test driver. How to deal with input data nondeterminism will be discussed further in Sect. 19.3.

Example 1 ([60]) Consider the concurrent C program shown in Fig. 2. This program represents a concurrent system composed of two processes that communicate using semaphores. The program describes the behavior of these processes as well as the initialization of the system. This example is inspired by the well-known dining-philosophers problem, with two philosophers. The two processes communicate by executing the (visible) operations *semwait* and *semsignal* on two semaphores that are identified by the integers 0 and 1 respectively. The operations *semwait* and

[3] This operation is called `VS_toss` in [60].

```
/* phil.c : dining philosophers (version without loops) */

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define N 2

philosopher(i,semid)
     int i, semid;
{
  printf("philosopher %d thinks\n",i);
  semwait(semid,i,1);          /* take left fork */
  semwait(semid,(i+1)%N,1);    /* take right fork */
  printf("philosopher %d eats\n",i);
  semsignal(semid,i,1);       /* release left fork */
  semsignal(semid,(i+1)%N,1); /* release right fork */
  exit(0);
}

main()
{
  int semid, i, pid;

  semid = semget(IPC_PRIVATE,N,0600);

  for(i=0;i<N;i++)
    semsetval(semid,i,1);
  for(i=0;i<(N-1);i++) {
    if((pid=fork()) == 0)
      philosopher(i,semid);
    };
  philosopher(i,semid);
}
```

Fig. 2 Example of concurrent C program simulating dining philosophers

semsignal take three arguments: the first argument is an identifier for an array of semaphores, the second is the index of a particular semaphore in that array, and the third argument is a value by which the counter associated with the semaphore identified by the first two arguments must be decremented (in the case of *semwait*) or incremented (in the case of *semsignal*). The value of both semaphores is initialized to 1 with the operation *semsetval*. By implementing these operations using actual operating-system semaphores (for instance, the exact UNIX system calls to do this are similar), the program above can be compiled and executed. The state space A_G of this system is shown in Fig. 3, where the two processes are denoted by $P1$ and $P2$, and state-space transitions are labeled with the visible operation of the corresponding process transition. The operation *exit* is a visible operation whose execution is always blocking. Since all the processes are deterministic, nondeterminism (i.e., branching) in A_G is caused only by concurrency. This state space contains two deadlocks (i.e., states with no outgoing transitions). The deadlock on the right represents normal termination (where both process are blocked on *exit*), while the deadlock on the left is due to a coordination problem.

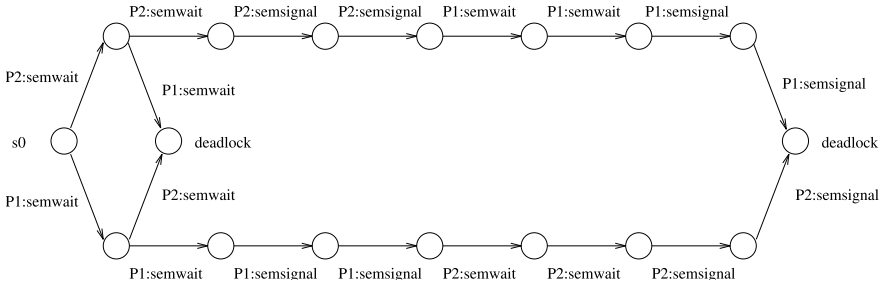


Fig. 3 Global state space for the two-dining-philosophers system

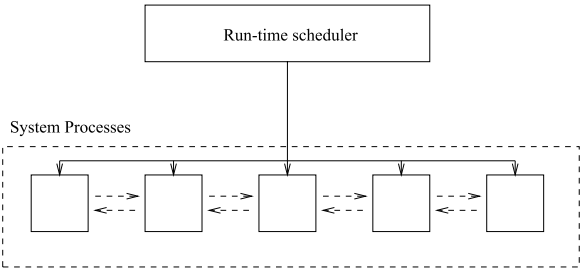


Fig. 4 Overall architecture of a dynamic software model checker for concurrent systems

19.2.3 Systematic Testing with a Run-Time Scheduler

The state space of a concurrent system as defined in the previous section can be systematically explored with a *run-time scheduler*. This scheduler controls and observes the execution of all the visible operations of the concurrent processes of the system (see Fig. 4). Every process of the concurrent system to be analyzed is mapped to an operating-system process. Their execution is controlled by the scheduler, which is another process external to the system. The scheduler observes the visible operations executed by processes inside the system, and can suspend their execution. By resuming the execution of (the next visible operation of) one selected system process in a global state, the scheduler can explore one transition in the state space A_G of the concurrent system.

Combined with a systematic state-space search algorithm, the run-time scheduler can drive an entire application through all (or many) possible concurrent executions by systematically scheduling all possible interleavings of their communication operations. In order to explore an alternative execution, i.e., to "backtrack" in its search, the run-time scheduler can, for instance, restart the execution of the entire software application in its initial state, and then drive its execution along a different path in its state space.

Whenever an error (such as a deadlock or an assertion violation) is detected during the search, a whole-system execution defined by the sequence of transitions that leads to the error state from the initial state can be exhibited to the user. Dynamic model checkers typically also include an interactive graphical simula-

tor/debugger for replaying executions and following their steps at the instruction or procedure/function level. Values of variables of each process can be examined interactively. The user can also interactively explore any path in the state space of the system with the same set of debugging tools (e.g., see [61]).

As described in the previous section, we assume that there are exactly two sources of nondeterminism in the concurrent systems considered here: (1) concurrency and (2) calls to the special visible operation `nondet` used to model nondeterminism and whose return values are controlled by the run-time scheduler. When this assumption is satisfied, the run-time scheduler has complete control over nondeterminism. It can thus reproduce any execution leading to an error found during a state-space search.

Remember that the ability to provide state-space coverage guarantees, even limited ones, is precisely what distinguishes verification, including model checking, from traditional testing, as explained earlier in the introduction. This is why the term "software model checking" was applied to this approach of systematic testing with a run-time scheduler, since eventually it does provide full state-space coverage.

Of course, in practice, state spaces can be huge, even infinite. But even then, the state space can always be explored exhaustively *up to some depth*, which can be increased progressively during state-space exploration using an "iterative deepening" search strategy. Efficient search algorithms, based on partial-order reduction, have been proposed for exhaustively exploring the state spaces of *message-passing* concurrent systems up to a "reasonable" depth, say, all executions with up to 50 message exchanges. In practice, such depths are often sufficient to thoroughly exercise implementations of communication protocols and other distributed algorithms. Indeed, exchanging a message is an expensive operation, and most protocols are therefore designed so that a few messages are sufficient to exercise most of their functionality. By being able to systematically explore all possible interactions of the implementation of all communicating protocol entities up to tens of message exchanges, this approach to software model checking has repeatedly been proven to be effective in revealing subtle concurrency-related bugs [61].

19.2.4 Stateless vs. Stateful Search

This approach to software model checking for concurrent programs thus adapts model checking into a form of systematic testing that simulates the effect of model checking while being applicable to concurrent processes executing arbitrary code written in full-fledged programming languages (like C, C++, Java, etc.). The only main requirement is that the run-time scheduler must be able to trap operating-system calls related to communication (such as sending or receiving messages) and be able to suspend and resume their executions, hence effectively controlling the scheduling of all processes whenever they attempt to communicate with each other.

This approach to software model checking was pioneered in the VeriSoft tool [60]. Because states of implementations of large concurrent software systems

can require megabytes or more *each* to be represented, VeriSoft does not store states in memory and simply traverse state-space paths in a *stateless* manner, exactly as in traditional testing. It is shown in [60] that in order to make a systematic stateless search tractable, partial-order reduction is necessary to avoid re-exploring over and over again parts of the state space reachable by different interleavings of the same concurrent partial-order execution.

However, for small to medium-size applications, computing state representations and storing visited states in memory can be tractable, possibly using approximations and especially if the entire state of the operating system can be determined, as is the case when the operating system is a *virtual machine*. This extension was first proposed in the Java PathFinder tool [148]. This approach limits the size and types of (here Java) programs that can be analyzed, but allows the use of standard model-checking techniques for dealing with state explosion, such as bitstate hashing, stateful partial-order reduction, and symmetry reduction, and the use of abstraction techniques.

Another trade-off is to store only *partial* state representations, such as storing a hash of a part of each visited state, possibly specified by the user, as explored in the CMC tool [108]. Full state-space coverage with respect to a dynamic semantics defined at the level of operating-system processes can then no longer be guaranteed, even up to some depth, but previously visited partial states can now be detected, and multiple explorations of their successor states can be avoided, which helps focus the remainder of the search on other parts of the state space more likely to contain bugs.

19.2.5 Systematic Testing for Multi-threaded Programs

Software model checking via systematic testing is effective for message-passing programs because systematically exploring their state spaces up to tens of message exchanges typically exercises a lot of their functionality. In contrast, this approach is more problematic for *shared-memory* programs, such as multi-threaded programs where concurrent threads communicate by reading and writing shared variables. Instead of a few well-identifiable message queues, shared-memory communication may involve thousands of communicating objects (e.g., memory addresses shared by different threads) that are hard to identify. Moreover, while systematically exploring all possible executions up to, say, 50 message exchanges can typically cover a large part of the functionality of a protocol implementation, systematically exploring all possible executions up to 50 read/write operations in a multi-threaded program typically covers only a tiny fraction of the program functionality. How to effectively perform software model checking via systematic testing for shared-memory systems is a harder problem and has been the topic of recent research.

Dynamic partial-order reduction (DPOR) [57] dynamically tracks interactions between concurrently executing threads in order to identify when communication takes place and through which shared variables (memory locations). Then, DPOR computes backtracking points where alternative paths in the state space need to be

explored because they might lead to other executions that are not "equivalent" to the current one (i.e., are not linearizations of the same partial-order execution). In contrast, traditional partial-order reduction [59, 118, 144] for shared-memory programs would require a static alias analysis to determine which threads may access which shared variables, which is hard to compute accurately and cheaply for programs with pointers. DPOR has been extended and implemented in several recent tools [82, 109, 135, 150].

Even with DPOR, state explosion is often still problematic. Another recent approach is to use *iterative context bounding*, a novel search-ordering heuristics which explores executions with at most k context switches, where k is a parameter that is iteratively increased [122]. The intuition behind this search heuristics is that many concurrency-related bugs in multi-threaded programs seem due to just a few unexpected context switches. This search strategy was first implemented in the Chess tool [109].

Even when prioritizing the search with aggressive context bounding, state explosion can still be brutal in large shared-memory multi-threaded programs. Other search heuristics for concurrency have been proposed, which we could collectively call *concurrency-fuzzing* techniques [19, 51, 132]. The idea is to use a random runtime scheduler that occasionally preempts concurrent executions selectively in order to increase the likelihood of triggering a concurrency-related bug in the program being tested. For instance, the execution of a memory allocation, such as `ptr=malloc(...)`, in one thread could be delayed as much as possible to see whether other threads may attempt to dereference that address `ptr` before it is allocated. Unlike DPOR or context bounding, these heuristic techniques do not provide any state-space coverage guarantees, but can still be effective in practice in finding concurrency-related bugs.

Other recent work investigates the use of concurrency-related search heuristics with *probabilistic guarantees* (e.g., see [19]). This line of work attempts to develop randomized algorithms for concurrent-system verification which can provide probabilistic coverage guarantees, under specific assumptions about the concurrent program being tested and for specific classes of bugs.

Active testing is a relatively new scalable automated method for directed testing of concurrent programs. Active testing combines the power of imprecise program analysis with the precision of software testing to quickly discover concurrency bugs and to reproduce discovered bugs on demand [22, 23, 25–27, 88–91, 115–117, 133]. The key idea behind active testing is to control the thread scheduler in order to force the program into a state that exposes a concurrency bug, e.g. data race, deadlock, atomicity violation, or violation of sequential memory consistency. The technique starts with lightweight inexpensive dynamic analysis that identifies situations where there is a suspicion that a concurrency bug may exist. This first part of the analysis is imprecise because it trades off precision for efficiency and tries to increase the coverage of analysis by trying to predict potential bugs in other executions by analyzing a single execution. In the second step, a directed tester executes the program under a controlled thread schedule in an attempt to bring the program into the buggy state. If it succeeds, it has identified a real concurrency bug; that is, the error

report is guaranteed not to be a false alarm, which is a serious problem with existing dynamic analyses. The actual method of controlling the thread schedule works as follows: once a thread reaches a state that resembles the desired state, it is paused as long as possible, giving a chance for other threads to catch up and complete the candidate buggy scenario.

19.2.6 Tools and Applications

We list here several tools and applications of software model checking via systematic testing for concurrent systems.

As mentioned before, the idea of dynamic software model checking via systematic testing was first proposed and implemented in the VeriSoft tool [60], developed at Bell Labs and publicly available since 1999. It has been used to find several complex errors in industrial communication software, ranging from small critical components of phone-switching software [64] to large call-processing applications running on wireless base-stations [34].

Java PathFinder [148] is another early and influential tool which analyzes concurrent Java programs using a modified Java virtual machine. It also implements a blend of several static and dynamic program analysis techniques. It has been used to find subtle errors in several complex Java components developed at NASA [17, 119]. It is currently available as an extensible open-source tool. It has recently been extended to also include test-generation techniques based on symbolic execution [2], which will be discussed in the next section.

CMC [108] analyzes concurrent C programs. It has been used to find errors in implementations of network protocols [107] and file systems [154].

jCUTE [135] is a tool for analyzing concurrent Java programs. It uses a variant of DPOR for data race detection. It also implements test-generation techniques based on concolic testing discussed in the next section.

Chess [109] analyzes multi-threaded Windows programs. It has been used to find many errors in a broad range of applications inside Microsoft [110]. It is also publicly available.

MaceMC [95] analyzes distributed systems implemented in Mace, a domain-specific language built on top of C++. This tool also specializes in finding liveness-related bugs.

MoDist [153] analyzes concurrent and distributed programs; it found many bugs in several distributed system implementations. Cuzz [19] analyzes multi-threaded Windows programs using concurrency-fuzzing techniques (see previous section). ISP [150] analyzes concurrent MPI programs using stateful variants of DPOR and other techniques.

Calfuzzer [89] is an extensible and publicly available active-testing tool for Java. Thrille for C/PThreads [88] and UPC-Thrille for UPC [116, 117] are active testing tools developed for C programs. These tools have been applied to find many previously known and unknown concurrency bugs in a number of programs, including

several real-world applications with millions of lines of code. UPC-Thrille is an active-testing tool for distributed-memory parallel programs. UPC-Thrille has been shown to scale to thousands of nodes with a maximum overhead of 50%.

19.3 Systematic Testing of Sequential Software

In this section, we present techniques inspired by model checking for systematically testing sequential software. We assume that nondeterminism in such programs is exclusively due to data input.

Enumerating all possible data inputs values with a `nondet` operation as described in Sect. 19.2.2 is tractable only when sets of possible input values are small, like selecting one choice in a menu with (few) options. For dealing with large sets of possible input data values, the main technical tool used is *symbolic execution*, which computes *equivalence classes of concrete input values* that lead to the execution of the same program path. We start with a brief overview of "classical" symbolic execution in the next section, and then describe recent extensions for systematic software testing.

19.3.1 Classical Symbolic Execution

Symbolic execution is a program analysis technique that was introduced in the 1970s (e.g., see [16, 41, 86, 96, 125]). Symbolic execution means executing a program with symbolic rather than concrete values. Assignment statements are represented as functions of their (symbolic) arguments, while conditional statements are expressed as constraints on symbolic values. Symbolic execution can be used for many purposes, such as bug detection, program verification, debugging, maintenance, and fault localization [42].

One of the earliest proposals for using static analysis as a kind of systematic symbolic program-testing method was made by King more than 35 years ago [96]. The idea is to symbolically explore the tree of all behaviors the program exhibits when all possible value assignments to input parameters are considered. For each *control path* ρ, that is, a sequence of control locations of the program, a *path constraint* ϕ_ρ is constructed that characterizes the input assignments for which the program executes along ρ. All the paths can be enumerated by a search algorithm that explores all possible branches at conditional statements. The paths ρ for which ϕ_ρ is satisfiable are *feasible* and are the only ones that can be executed by the actual program. The solutions to ϕ_ρ characterize the inputs that drive the program through ρ. This characterization is exact provided symbolic execution has perfect precision. Assuming that the theorem prover used to check the satisfiability of all formulas ϕ_ρ is sound and complete, this use of static analysis amounts to a kind of symbolic testing. How to perform symbolic execution and generate path constraints is illustrated with an example later in Sect. 19.3.4.

A prototype of this system allowed the programmer to be presented with feasible paths and to experiment, possibly interactively [78], with assertions in order to force new and perhaps unexpected paths. King noticed that assumptions, now called preconditions, also formulated in the logic could be joined to the analysis, forming, at least in principle, an automated theorem prover for Floyd–Hoare's verification method [58, 83], including inductive invariants for programs that contain loops. Since then, this line of work has been developed further in various ways, leading to various approaches to program verification, such as *verification-condition generation* (e.g., [5, 48]), *symbolic model checking* [18] and *bounded model checking* [37].

Symbolic execution is also a key ingredient for *precise test input generation* and systematic testing of *data-driven programs*. While program verification aims at proving the absence of program errors, test generation aims at generating concrete test inputs that can drive the program to execute specific program statements or paths. Work on automatic code-driven test generation using symbolic execution can roughly be partitioned into two groups: *static* versus *dynamic* test generation.

19.3.2 Static Test Generation

Static test generation (e.g., [96]) consists of analyzing a program P statically, by using symbolic execution techniques to attempt to compute inputs to drive P along specific execution paths or branches, *without ever executing the program*.

Unfortunately, this approach is ineffective whenever the program contains statements involving constraints outside the scope of reasoning of the theorem prover, i.e., statements "that cannot be reasoned about symbolically". This limitation is illustrated by the following example [62]:

```
int obscure(int x, int y) {
  if (x == hash(y)) abort();      // error
  return 0;                       // ok
}
```

Assume the constraint solver cannot "symbolically reason" about the function `hash` (perhaps because it is too complex or simply because its code is not available). This means that the constraint solver cannot generate two values for inputs `x` and `y` that are guaranteed to satisfy (or violate) the constraint `x == hash(y)`. In this case, static test generation cannot generate test inputs to drive the execution of the program `obscure` through either branch of the conditional statement: static test generation is *helpless* for a program like this. Note that, for test generation, it is not sufficient to know that the constraint `x == hash(y)` is satisfiable for *some* values of `x` and `y`, it is also necessary to generate *specific values* for `x` and `y` that satisfy or violate this constraint.

The practical implication of this fundamental limitation is significant: static test generation is doomed to perform poorly whenever precise symbolic execution is not

possible. Unfortunately, this is frequent in practice due to complex program statements (pointer manipulations, floating-point operations, etc.) and calls to operating-system and library functions that are hard or impossible to reason about symbolically with good enough precision.

19.3.3 Dynamic Test Generation

A second approach to test generation is *dynamic test generation* (e.g., [29, 68, 80, 97, 114]): it consists of executing the program P, typically starting with some random inputs, while performing symbolic execution *dynamically*, collecting symbolic constraints on inputs gathered from predicates in branch statements along the execution, and then using a constraint solver to infer variants of the previous inputs in order to steer the next execution of the program towards an alternative program branch. The conventional stance on the role of symbolic execution is thus turned upside-down: symbolic execution is now an adjunct to concrete execution.

A key observation [68] is that, with dynamic test generation, *imprecision in symbolic execution can be alleviated using concrete values and randomization*: whenever symbolic execution does not know how to generate a constraint for a program statement depending on some inputs, one can always simplify this constraint using the concrete values of those inputs.

Consider again the program `obscure` given above. Even though it is impossible to generate two values for inputs `x` and `y` such that the constraint `x == hash(y)` is satisfied (or violated), it is easy to generate, for a fixed value of `y`, a value of `x` that is equal to `hash(y)` since the latter can be observed and known at run-time. By picking randomly and then fixing the value of `y`, we can first run the program, observe the concrete value c of `hash(y)` for the fixed value of `y` in that run; then, in the next run, we can set the value of the other input `x` either to c or to another value, while leaving the value of `y` unchanged, in order to force the execution of the `then` or `else` branches, respectively, of the conditional statement in the function `obscure`. (The algorithm presented in the next section does all this automatically [68].)

In other words, static test generation is unable to generate test inputs to control the execution of the program `obscure`, while dynamic test generation can *easily* drive the executions of that same program through all its feasible program paths, finding the `abort()` with no false alarms. In realistic programs, imprecision in symbolic execution typically creeps in in many places, and dynamic test generation allows test generation to recover from that imprecision. Dynamic test generation can be viewed as extending static test generation with additional run-time information, and is therefore more general, precise, and powerful.

How much more precise is dynamic test generation compared to static test generation? In [63], it is shown exactly when the "concretization trick" used in the above `obscure` example helps, and when it does not help. It is also shown that the main property of dynamic test generation that makes it more powerful than static

test generation is *only* its ability to observe concrete values and to record them in path constraints. In contrast, the process of simplifying complex symbolic expressions using concrete run-time values can be accurately simulated statically using *uninterpreted functions*. However, those concrete values are necessary to effectively compute new input vectors, a fundamental requirement in test generation [63].

In principle, static test generation can be extended to concretize symbolic values whenever static symbolic execution becomes imprecise [94]. In practice, this is problematic and expensive because this approach not only requires the detection of *all* sources of imprecision, but also requires one call to the constraint solver for each concretization to ensure that every synthesized concrete value satisfies prior symbolic constraints along the current program path. In contrast, dynamic test generation avoids these two limitations by leveraging a specific concrete execution as an automatic fallback for symbolic execution [68].

In summary, dynamic test generation is the *most precise* form of code-driven test generation that is known today. It is more precise than static test generation and other forms of test generation such as random, taint-based, and coverage-heuristic-based test generation. It is also the most sophisticated, requiring the use of automated theorem proving for solving path constraints. This machinery is more complex and heavyweight, but may exercise more paths, find more bugs and generate fewer redundant tests covering the same path. Whether this better precision is worth the trouble depends on the application domain.

19.3.4 Systematic Dynamic Test Generation

Dynamic test generation was discussed in the 1990s (e.g., [80, 97, 114]) in a *property-guided* setting, where the goal is to execute a given *specific target* program branch or statement. More recently, new variants of dynamic test generation [29, 68] blend it with model-checking techniques with the goal of systematically executing *all* feasible program paths of a program while detecting various types of errors using run-time checking tools (like Purify, Valgrind, or AppVerifier, for instance). In other words, each new input vector attempts to force the execution of the program through *some* new path, but the whole search is *not* guided by one specific target program branch or statement. By repeating this process, such a systematic search attempts to force the program to sweep through all its feasible execution paths, in a style similar to *systematic testing* and *dynamic software model checking* [60] as presented in Sect. 19.2. Along each execution, a run-time checker is used to detect various types of errors (buffer overflows, uninitialized variables, memory leaks, etc.).

Systematic dynamic test generation as described above was first introduced in [68], as a part of an algorithm for "Directed Automated Random Testing", or DART for short, and is also referred to as "concolic testing" [138], or "dynamic symbolic execution" [143]. Independently, [29] proposed "Execution-Generated Tests" as a test-generation technique very similar to DART. Also independently, [151] described a prototype tool which shares some of the same features. Subsequently, this

approach was adopted and implemented in many other tools (see Sect. 19.3.6 and surveys [31, 32]).

Systematic dynamic test generation consists of running the program P under test both concretely, executing the actual program, and symbolically, calculating constraints on values stored in program variables v and expressed in terms of input parameters. Side-by-side concrete and symbolic executions are performed using a concrete store M and a symbolic store S, which are mappings from *memory addresses* (where program variables are stored) to concrete and symbolic values respectively [68]. A *symbolic value* is any expression e in some theory[4] $\mathscr{T}$ where all free variables are exclusively input parameters. For any program variable v, $M(v)$ denotes the *concrete value* of v in M, while $S(v)$ denotes the *symbolic value* of v in S. For notational convenience, we assume that $S(v)$ is always defined and is simply $M(v)$ by default if no symbolic expression in terms of inputs is associated with v in S. When $S(v)$ is different from $M(v)$, we say that that program variable v has a *symbolic value*, meaning that the value of program variable v is a function of some input(s) which is represented by the symbolic expression $S(v)$ associated with v in the symbolic store.

A program manipulates the memory (concrete and symbolic stores) through statements, or *commands*, which are abstractions of the machine instructions actually executed. A command can be an *assignment* of the form $v := e$ where v is a program variable and e is an expression, a *conditional statement* of the form `if` b `then` C' `else` C'' where b denotes a boolean expression, and C' and C'' denote the unique[5] next command to be evaluated when b holds or does not hold, respectively, or `stop` corresponding to a program error or normal termination.

Given an input vector assigning a concrete value to every input parameter I_i, the program executes a unique finite[6] sequence of commands. For a finite sequence ρ of commands (i.e., a control path ρ), a *path constraint* ϕ_ρ is a *quantifier-free first-order logic formula* over theory $\mathscr{T}$ that is meant to characterize the input assignments for which the program executes along ρ. The path constraint is *sound and complete* when this characterization is exact.

A path constraint is generated during dynamic symbolic execution by collecting input constraints at conditional statements. Initially, the path constraint ϕ_ρ is defined as $true$, and the initial symbolic store S_0 maps every program variable v whose initial value is a program input: for all these, we have $S_0(v) = x_i$ where x_i is the symbolic variable corresponding to the input parameter I_i. During dynamic symbolic execution, whenever an assignment statement $v := e$ is executed, the symbolic store is updated so that $S(v) = \sigma(e)$ where $\sigma(e)$ either denotes an expression in $\mathscr{T}$ representing e as a function of its symbolic arguments, or is simply the current concrete value $M(v)$ of v if e does not have symbolic arguments or if

[4] A theory is a set of logic formulas.

[5] We assume in this section that program executions are sequential and deterministic.

[6] We assume program executions terminate. In practice, a timeout can prevent non-terminating program executions and issue a run-time error.

```
int f(int x) { return 2 * x; }
int h(int x, int y) {
  if (x != y)
     if (f(x) == x + 10)
        abort();     // error
  return 0;
}
```

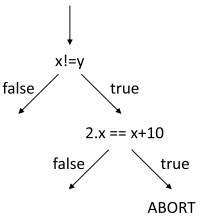


Fig. 5 A sample program (*left*) and the tree formed by all its path constraints (*right*)

e cannot be represented by an expression in $\mathscr{T}$. Whenever a conditional statement `if` b `then` C' `else` C'' is executed and the `then` (respectively `else`) branch is taken, the current path constraint ϕ_ρ is updated to become $\phi_\rho \wedge c$ (respectively $\phi_\rho \wedge \neg c$) where $c = \sigma(b)$. Note that, by construction, all symbolic variables ever appearing in ϕ_ρ are variables x_i corresponding to whole-program inputs I_i.

Given a path constraint $\phi_\rho = \bigwedge_{1 \le i \le n} c_i$, new alternate path constraints ϕ'_ρ can be defined by negating one of the constraints c_i and putting it in a conjunction with all the previous constraints: $\phi'_\rho = \neg c_i \wedge \bigwedge_{1 \le j < i} c_j$. If path constraint generation is sound and complete, any satisfying assignment to ϕ'_ρ defines a new test input vector which will drive the execution of the program along the same control-flow path up to the conditional statement corresponding to c_i where the new execution will then take the other branch. By systematically repeating this process, such a *directed search* can enumerate all possible path constraints and eventually execute all feasible program paths.

The search is exhaustive provided that the generation of the path constraint (including the underlying symbolic execution) and the constraint solver for the given theory $\mathscr{T}$ are both *sound and complete*, that is, for all program paths ρ, the constraint solver returns a satisfying assignment for the path constraint ϕ_ρ *if and only if* the path is feasible (i.e., there exists some input assignment leading to its execution). If those conditions hold, in addition to finding errors such as the reachability of bad program statements (like `abort()` or `assert(false)`), a directed search can also prove their absence, and therefore perform a form of program *verification*.

In practice, path constraint generation and constraint solving are usually not sound and complete. Moreover, in the presence of a single loop whose number of iterations depends on some unbounded input, the number of feasible program paths becomes infinite. In practice, search termination can always be forced by bounding input values, loop iterations, or recursion, but at the cost of potentially missing bugs.

Example 2 ([68]) Consider the function `h` shown in Fig. 5. The function `h` is defective because it may lead to an abort statement for some value of its input vector, which consists of the input parameters `x` and `y`. Running the program with random values for `x` and `y` is unlikely to discover the bug.

Assume we start with some random initial concrete input values, say `x` is initially 269167349 and `y` is 889801541. Initially, every program input is associated

with a symbolic variable, denoted respectively by x and y, and every program variable storing an input value has its symbolic value (in the symbolic store) associated with the symbolic variable for the corresponding input: thus, the symbolic value for program variable `x` is the symbolic value x, and so on. Initially, the path constraint is simply $true$.

Running the function `h` with these two concrete input values executes the `then` branch of the first if-statement, but fails to execute the `then` branch of the second if-statement; thus, no error is encountered. This execution defines a path ρ through the program. Intertwined with the normal execution, dynamic symbolic execution generates the path constraint $\phi_\rho = (x \neq y) \wedge (2 \cdot x \neq x + 10)$. Note the expression $2 \cdot x$, representing `f(x)`: it is defined through an interprocedural, dynamic tracing of symbolic expressions.

The path constraint ϕ_ρ represents an equivalence class of input vectors, namely all the input vectors that drive the program through the path that was just executed. To force the program through a different equivalence class, the directed search generates the new path constraint, say, $\phi'_\rho = (x \neq y) \wedge (2 \cdot x = x + 10)$ obtained by negating the last constraint of the current path constraint (for instance, if the search is performed in a depth-first order). A solution to this new path constraint is $(x = 10, y = 889801541)$. A second execution of the function `h` with these two new input values then reveals the error by driving the program into the `abort()` statement as expected.

The search space to be explored for this program is shown to the right of Fig. 5. Each path in this tree corresponds to a path constraint. When symbolic execution has perfect precision as in this simple example, path constraints are both sound and complete, and dynamic and static test generation are equally powerful: they can both generate tests to drive the program along all its execution paths.

Example 3 Consider again the function `obscure`:

```
int obscure(int x, int y) {
  if (x == hash(y)) abort();      // error
  return 0;                       // ok
}
```

Assume we start running this program with some initial random concrete values, say `x` is initially 33 and `y` is 42. During dynamic symbolic execution, when the conditional statement is encountered, assume we do not know how to represent the expression `hash(y)`. However, we can observe dynamically that the concrete value of `hash(42)` is, say, 567. Then, the *simplified path constraint* $\phi_\rho = (x \neq 567)$ can be generated by replacing the complex/unknown symbolic expression `hash(y)` by its concrete value 567. This constraint is then negated and solved, leading to the new input vector $(x = 567, y = 42)$. Running the function `obscure` a second time with this new input vector leads to the `abort()` statement. When symbolic execution does *not* have perfect precision, dynamic test generation can be more precise than static test generation as illustrated with this example since dynamic test generation is still able to drive this program along all its feasible paths, while static test generation cannot.

Fig. 6 Another program example with C-like syntax

```
struct foo {int i; char c;}
bar (char x) {
  struct foo *a;
  a->c = x;
  if (a->c == 0) {
     *((char *)a + sizeof(int)) = 1;
     if (a->c != 0)
        abort();
  }
}
```

Example 4 (adapted from [68]) Consider the C-like program shown in Fig. 6. For this example, a static analysis will typically not be able to report with high certainty that the `abort()` is reachable. Sound static analysis tools will report "the abort might be reachable", and unsound ones will simply report "no bug found", if their alias analysis is not able to guarantee that `a->c` has been overwritten. In contrast, dynamic test generation easily finds a precise execution leading to the abort by simply generating an input satisfying the constraint $x = 0$. Indeed, the complex pointer arithmetic expression `*((char *)a + sizeof(int)) = 1` is *not input-dependent*, and its symbolic execution is therefore reduced to a purely concrete execution where the left-hand side of the assignment is mapped to a single concrete address—no symbolic pointer arithmetic is required, nor any pointer alias analysis. This kind of code is often found in implementations of network protocols, when a buffer of type `char *` representing an incoming message is cast into a `struct` representing the different fields of the message type.

19.3.5 Strengths and Limitations

At a high level, systematic dynamic test generation suffers from two main limitations:

1. the frequent imprecision of symbolic execution along individual paths, and
2. the large number of paths that usually need be explored, or *path explosion*.

In practice, however, approximate solutions to the two problems above are sufficient. To be useful, symbolic execution does not need to be perfect, it must simply be "good enough" to drive the program under test through program branches, statements and paths that would be difficult to exercise with simpler techniques like random testing. Even if a directed search cannot typically explore all the feasible paths of large programs in a reasonable amount of time, it usually does achieve better coverage than pure random testing and, hence, can find new program bugs.

Another key advantage of dynamic symbolic execution is that it can be implemented *incrementally*: only some program statements can be instrumented and interpreted symbolically, while others can simply be executed concretely natively, including all calls to external libraries and operating-system functions. A tool developer can improve the precision of symbolic execution over time, by adding new instruction handlers in a modular manner. Similarly, simple techniques like bounding

the number of constraints injected at each program location are effective practical solutions to limit path explosion.

When building tools like these, there are many other challenges, which have recently been discussed in the research literature: how to recover from imprecision in symbolic execution [63, 68, 138], how to scale symbolic execution to billions of instructions [71], how to efficiently check many properties together [30, 71, 92], how to automatically synthesize symbolic instruction handlers [76], how to infer data structure invariants [93], how to reason about pointers [30, 52, 138], how to combine with random testing [101], how to find algorithmic performance problems [21], how to detect infinite loops in running programs [20], how to deal with inputs of varying sizes [152], how to deal with floating-point instructions [67], how to deal with path explosion using compositional test summaries and other caching techniques [1, 14, 62, 75, 103], which heuristics to use to prioritize the search in the program's search space [24, 30, 72], how to deal specifically with input-dependent loops [74, 129], how to leverage grammars (when available) for complex input formats [66, 102], how to reuse previous analysis results across code changes [70, 120, 121], how to leverage reachability facts inferred by static program analysis [75], etc. Due to space constraints, we do not discuss these challenges here, but instead refer the reader to the recent references above where these problems are discussed in detail and more pointers to other related work are provided.

19.3.6 Tools and Applications

Despite the limitations and challenges mentioned in the previous section, systematic dynamic test-generation works well in practice: it is often able to detect bugs missed by other less precise test generation techniques. Moreover, because it is grounded in concrete executions, this approach does not report false alarms, unlike traditional static program analysis. These strengths explain the popularity of the approach and its adoption in many recent tools.

Over the last several years, several tools implementing dynamic test generation have been developed for various programming languages, properties, and application domains. Examples of such tools are DART [68], EGT [29], PathCrawler [151], CUTE [138], EXE [30], SAGE [72], CatchConv [105], PEX [143], KLEE [28], CREST [24], BitBlaze [140], Splat [103], Apollo [3], YOGI [75], Kudzu [128], S2E [36], CATG [130], and Jalangi [137], among others.

The above tools differ by how they perform dynamic symbolic execution (for languages such as C, Java, x86, .NET, etc.), by the type of constraints they generate (for theories such as linear arithmetic, bit-vectors, arrays, uninterpreted functions, etc.), and by the type of constraint solvers they use (such as lp_solve, CVClite, STP, Disolver, Yices, Z3, etc.). Indeed, like in traditional static program analysis and abstract interpretation, these important parameters are determined in practice depending on which type of program is to be tested, on how the program interfaces with its environment, and on which properties are to be checked. Moreover, various cost/precision tradeoffs are also possible, as usual in program analysis.

The tools listed above also differ by the specific application domain they target, for instance protocol security [68], Unix utility programs [28, 30], database applications [53], Web applications [3, 128, 137], and device drivers [75, 98]. The size of the software applications being tested also varies widely, from unit testing of programs [24, 30, 36, 68, 137, 138, 143] to system testing of very large programs with millions of lines of code [71].

At the time of writing, the largest-scale usage and deployment of systematic dynamic test-generation is for *whitebox fuzzing of file parsers* [72], i.e., whole-application testing for security vulnerabilities (buffer overflows, etc.). Whitebox fuzzing scales to large file parsers embedded in applications with millions of lines of code, such as Excel, and execution traces with billions of machine instructions. Whitebox fuzzing was first implemented in the tool SAGE [72], which uses the Z3 [106] *Satisfiability-Modulo-Theories* (SMT) solver as its constraint solver. Since 2008, SAGE has been running for over 500 machine years in Microsoft's security testing labs. This currently represents the largest computational usage for any SMT solver, with billions of constraints processed to date [15]. In the process, SAGE found new security vulnerabilities in hundreds of applications, including image processors, media players, file decoders, and document parsers. Notably, SAGE found roughly one third of all the bugs discovered by file fuzzing during the development of Microsoft's Windows 7, saving millions of dollars by avoiding expensive security patches for a billion PCs [73].

19.4 Systematic Testing of Concurrent Software with Data Inputs

Dynamic test generation for sequential software assumes that the program under test has no nondeterminism due to concurrency. Real-world programs have data inputs and are almost invariably concurrent. jCUTE, a dynamic test generation technique [131, 134–136], shows how to systematically test programs that have nondeterminism due to both concurrency and data inputs. The technique combines a variant of dynamic partial-order reduction (see Sect. 19.2.5), called the race-detection and flipping algorithm, with concolic testing. The goal of the technique is to generate thread schedules as well as data inputs that exercise all non-equivalent execution paths of a shared-memory multi-threaded program. The technique has been implemented for Java programs in the open-source tool jCUTE (available at http://osl.cs.illinois.edu/software/jcute/).

jCUTE works as follows. Like DART, jCUTE executes a program both concretely and symbolically. Along the concrete execution path, jCUTE collects the constraints over the symbolic input values at each branch point and computes the path constraint. Apart from collecting symbolic constraints, jCUTE also computes race conditions (both data races and lock races) between various events in the execution of a program, where, informally, an event represents the execution of a statement in the program by a thread. We say that two events are in a *race* if they are

```
x is a shared variable
z = input();

Thread t1

1:  x = 3;

Thread t2

1:  x = 2;
2:  if (2*z + 1 == x)
3:      abort();
```

Fig. 7 A simple shared-memory multi-threaded program P

events of different threads, they access (i.e. read, write, lock, or unlock) the same memory location without holding a common lock, and the order of occurrence of the events can be permuted by changing the schedule of the threads. The race conditions are computed by analyzing the concrete execution of the program with the help of a standard dynamic vector clock algorithm.

jCUTE first generates a random input and a schedule, which specifies the order of execution of threads. Then the algorithm does the following in a loop: it executes the code with the generated input and schedule. Along the execution path the algorithm computes the symbolic path constraint as well as the race conditions between various events. It backtracks and generates a new schedule or a new input and executes the program again. It continues until it has explored all possible distinct execution paths using a depth-first search strategy. The choice of new inputs and schedules is made in one of the following two ways:

1. The algorithm picks a constraint from the symbolic constraints that were collected along the execution path and negates the constraint to define a new path constraint. The algorithm then finds, if possible, some concrete values that satisfy the new path constraint. These values are used as inputs for the next execution.
2. The algorithm picks two events which are in a race and generates a new schedule where the execution of the thread involved in the first event is *postponed* or *delayed* as much as possible right before the event occurs. This ensures that the events involved in the race get *flipped* or re-ordered when the program is executed with the new schedule. The new schedule is used for the next execution.

19.4.1 Example

We illustrate how jCUTE performs concolic testing along with race-detection and flipping using the sample program P in Fig. 7. The program has two threads t_1 and t_2, a shared integer variable `x`, and an integer variable `z` which receives an input from the external environment at the beginning of the program. Each statement in the program is labeled. The program reaches the `abort()` statement in thread t_2 if the input to the program is `1` (i.e., `z` gets the value `1`) and the program executes the statements in the following order: $(t_2, 1)(t_1, 1)(t_2, 2)(t_2, 3)$, where each event (t, l) in the sequence denotes that the thread t executes the statement labeled l.

jCUTE first generates a random input for z and executes P with a default schedule. Without loss of generality, the default schedule always picks the thread which is enabled and which has the lowest index. Thus the first execution of P is $(t_1, 1)(t_2, 1)(t_2, 2)$. Let z_0 be the symbolic value of z at the beginning of the execution. jCUTE collects the constraints from the predicates of the branches executed in this path. For this execution, jCUTE generates the path constraint $\langle 2 * z_0 + 1! = 2\rangle$. jCUTE also discovers that there is a race condition between the first and the second event because both the events access the same variable x in different threads without holding a common lock and one of the accesses is a write of x.

Following the depth-first search strategy, jCUTE picks the only constraint $2 * z_0 + 1! = 2$, negates it, and tries to solve the negated constraint $2 * z_0 + 1 = 2$. This has no solution. Therefore, jCUTE backtracks and generates a schedule such that the next execution becomes $(t_2, 1)(t_2, 2)(t_1, 1)$. In this execution the thread involved in the first event of the race in the previous execution is delayed as much as possible. The execution thus re-orders the events involved in the race in the previous execution.

During the above execution, jCUTE generates the path constraint $\langle 2 * z_0 + 1! = 2\rangle$ and discovers that there is a race between the second and the third events. Since the negated constraint $2 * z_0 + 1 = 2$ cannot be solved, jCUTE backtracks and generates a schedule such that the next execution becomes $(t_2, 1)(t_1, 1)(t_2, 2)$. This execution re-orders the events involved in the race in the previous execution.

In the above execution, jCUTE generates the path constraint $\langle 2 * z_0 + 1! = 3\rangle$. jCUTE solves the negated constraint $2 * z_0 + 1 = 3$ to obtain $z_0 = 1$. In the next execution, it follows the same schedule as the previous execution. However, jCUTE starts the execution with the input variable z set to 1 which is the value of z that jCUTE computed by solving the constraint. The resultant execution becomes $(t_2, 1)(t_1, 1)(t_2, 2)(t_2, 3)$ which hits the `abort()` statement of the program.

For each data input, the algorithm in jCUTE explores all thread schedules that are not "equivalent" to each other (i.e., are not linearizations of the same partial-order execution). Proof of this correctness result can be found in [131].

19.4.2 Comparison with Related Work

The race-detection and flipping algorithm developed in jCUTE is a variant of dynamic partial-order reduction. The key difference between the DPOR algorithm [57] and our race-detection and flipping algorithm is that, for every choice point, the DPOR algorithm uses a persistent set while we use a postponed set [136]. These two sets can be different at a choice point. For example, for the three-threaded program in Fig. 8, if the first execution path is $(t_1, 1)(t_2, 2)(t_3, 3)$, then at the first choice point denoting the initial state of the program, the persistent set is $\{t_1, t_3\}$; whereas, at the same choice point, the postponed set is $\{t_1\}$. (Apart from scheduling the thread t_1, the race-detection and flipping algorithm also schedules the thread t_2 at the first choice point.) Note that the DPOR algorithm picks the elements of a persistent set by using a complex forward lookup algorithm. In contrast, jCUTE *simply* puts the current scheduled thread to the postponed set at a choice point.

```
t1:              t2:              t3:
1: x = 1;        2: y = 4;        3: x = 2;
```

Fig. 8 A three-threaded program

Note that none of the previous descriptions of DPOR techniques handled programs with data inputs. It is also worth noting that the race-detection and flipping algorithm is dynamic in nature like DPOR and addresses the limitations of static partial-order reduction [59, 118, 144].

In [134] concolic testing has been extended to test asynchronous message-passing Java programs written using a Java Actor library. Shared memory systems can be modeled as asynchronous message-passing systems by associating a thread with every memory location. Reads and writes of a memory location can be modeled as asynchronous messages to the thread associated with the memory location. However, this particular model would treat both reads and writes similarly. Hence, the algorithm in [134] would explore many redundant executions. For example, for the two-threaded program $t_1 : x = 1; x = 2; t_2 : y = 3; x = 4;$, the algorithm in [134] would explore six interleavings. Our algorithm assumes that two reads are not in a race and thus would explore only three interleavings of the program.

In a similar independent work [139], Siegel et al. use a combination of symbolic execution and static partial-order reduction to check whether a parallel numerical program is equivalent to a simpler sequential version of the program. However, their main emphasis is in symbolic execution of numerical programs with floating points, rather than programs with pointers and data structures. Therefore, static partial-order reduction proves effective in their approach.

Model checking tools [45, 141] based on static analysis have been developed that can detect bugs in concurrent programs. These tools employ partial-order reduction techniques to reduce search space. The partial-order reduction depends on detection of thread-local memory locations and patterns of lock acquisition and release.

The Path Exploration Tool (PET) [79] allows the (static) symbolic calculation of path conditions of sequential as well as concurrent systems. One can interactively change the path (for concurrent systems this can include swapping the order of concurrent transitions) to test the effect on the execution. In particular, it is shown how to limit the path condition based on a temporal property that the path needs to satisfy, represented using an automaton. In order to extend the application of symbolic testing and verification, calculating path conditions for concurrent systems with time constraints was presented in [7] and symbolic calculation of path conditions for infinite (ultimately periodic) paths was proposed in [8]. These two extensions were integrated into the PET system.

A recent related work [126] uses a set of program runs as opposed to the actual program along with concolic testing to increase coverage to concurrent program testing. Test generation is done by solving a constraint system that encodes the scheduling constraints and the data-flow constraints together. The technique is incomplete because it analyzes a subset of program traces.

More recently Farzan et al. [55] propose an alternative technique for testing concurrent programs. The technique uses interference scenarios to systematically

explore the execution space of a concurrent program. Interference scenarios capture the flow of data among different threads and enable a unified representation of path and interference constraints. The technique has been shown to scale better than jCUTE.

19.5 Other Related Work

The techniques we presented for software model checking by systematic testing for concurrency (Sect. 19.2) and for data inputs (Sect. 19.3) can be combined and used together (Sect. 19.4). Indeed, nondeterminism due to concurrency (whom to schedule) is *orthogonal* to nondeterminism due to input data (what values to provide). For checking most properties, concurrent programs can be sequentialized using an interleaving semantics (e.g., [60, 123]). Therefore, symbolic execution can be extended to multi-threaded programs [2, 135], since threads share the same memory address space, and take advantage of partial-order reduction (e.g., [57, 59]). The case of multi-process programs is more complicated since a good solution requires tracking symbolic variables across process boundaries and through operating-systems objects such as message queues.

Static Abstraction-Based Software Model Checking. As mentioned in the introduction, essentially two approaches to software model checking have been proposed and are still actively investigated. The first approach is the one presented in the previous sections. The second approach consists of automatically extracting a model out of a software application by statically analyzing its code and abstracting away details, applying traditional model checking to analyze this abstract model, and then mapping abstract counterexamples (if any) back to the code. The investigation of this *abstraction-based* second approach can be traced back to early attempts to analyze concurrent programs written in concurrent programming languages such as Ada (e.g., [44, 100, 104, 142]). Other relevant work includes static analyses geared towards analyzing communication patterns in concurrent programs (e.g., [43, 46, 147]). Starting in the late 1990s, several efforts have aimed at providing model-checking tools based on source-code abstraction for mainstream popular programming languages such as C and Java. For instance, Feaver [85] can translate C programs into Promela, the input language of the SPIN model checker, using user-specified abstraction rules. Similarly, Bandera [45] can translate Java programs into the (finite-state) input languages of existing model checkers like SMV and SPIN, using user-guided abstraction, slicing, and abstract interpretation techniques. The abstraction process can also be made *fully automatic* and adjustable depending on the specific property to be checked. For instance, SLAM [4] can translate sequential C programs to "boolean programs", which are essentially inter-procedural control-flow graphs extended with boolean variables, using an *automatic* iterative abstraction-refinement process based on the use of predicate abstraction and a specialized model-checking procedure. For the specific classes of programs that these tools can handle, the use of abstraction techniques can produce a "conservative"

model of a program that preserves basic information about the execution and communication patterns taking place in the system executing the program. Analyzing such a model using standard model-checking techniques can then prove the absence of certain types of errors in the system, without ever executing the program itself.

This second approach of static software model checking via abstraction is *complementary* to dynamic software model checking via systematic testing. Both approaches inherit the well-known advantages and limitations of, respectively, static and dynamic program analysis (e.g., [54]). Static analysis is faster than testing, provides better code coverage, but is usually less precise, is language dependent, and may produce spurious counterexamples (i.e, suffers from "false alarms/positives"). In contrast, dynamic analysis is precise, more language-independent, detects real bugs, but is slower, provides usually less coverage, and can miss bugs (i.e., suffers from "false negatives"). Overall, static and dynamic program analysis have *complementary strengths and weaknesses*, and are *worth combining*.

Combining Static and Dynamic Software Model Checking. There are many ways to combine static and dynamic program analysis, and, similarly, to combine static and dynamic software model checking. Several algorithms and tools combine static and dynamic program analyses for property checking and test generation, e.g., [9–12, 47, 112, 149]. Most of these loose combinations perform a static analysis before a dynamic analysis, while some [10–12] allow for some feedback to flow between the two. A tight integration between static and dynamic software model checking is proposed in a series of algorithms named Synergy [77], Dash [6] and Smash [75], and implemented in the Yogi tool [113]. The latest of these algorithms performs a compositional interprocedural may/must program analysis, where two complementary sets of techniques are used and intertwined together: a may analysis for finding proofs based on predicate abstraction and automatic abstraction refinement as in SLAM [4], and a must analysis for finding bugs based on dynamic test generation as in DART [68]. These two analyses are performed together, in coordination, and communicate their respective intermediate results to each other in the form of reusable may and must procedure summaries. This fined-grained coupling between may and must summaries allows the flexible and demand-driven use of either type of summaries for both proving and disproving program properties in a sound manner, and was shown experimentally to outperform previous algorithms of this kind for property-guided verification [75].

Run-Time Verification. In contrast, the approach taken in systematic dynamic test generation (see Sect. 19.3.4) is *not* property-guided: the goal is instead to exercise as many program paths as possible while checking *many* properties simultaneously along each of those paths [71]. In a non-property-guided setting, the effectiveness of static analysis for safely cutting parts of the search space is typically more limited. In this context, other complementary work includes tools like Purify, Valgrind and AppVerifier, that automatically instrument code or executable files for monitoring program executions and detecting at run-time standard programming and memory-management errors such as array out-of-bounds and memory leaks. Also, several tools for so-called *run-time verification* that monitor the behavior of a reactive program at run-time and compare this behavior against an application-specific high-level specification (typically a finite-state automaton or a temporal

logic formula) have recently been developed (e.g., [50, 81]). These tools can also be used in conjunction with dynamic software model checkers.

Model-Based Testing. Software model checking via systematic testing differs from *model-based testing*. Given an abstract representation of the program, called a *model*, model-based testing consists of generating tests to check the *conformance* of the program with respect to the model (e.g., [13, 35, 49, 87, 127, 146, 155]). In contrast, systematic testing does not use or require a model of the program under test. Instead, its goal is to generate tests that exercise as many program statements as possible, including assertions inserted in the code. Another fundamental difference is that models are usually written in abstract modeling languages which are, by definition, more amenable to precise analysis, symbolic execution, and test generation. In contrast, code-driven test generation has to deal with arbitrary software code and systems for which program analysis is bound to be imprecise, as we discussed in Sects. 19.3.2 and 19.3.3. Sometimes, the model itself is specified as a product of finite-state machines (e.g., [56]). In that case, systematic state-space exploration techniques inspired by traditional finite-state model checking are used to automatically generate a set of test sequences that cover the concurrent model according to various coverage criteria.

Must Program Abstractions. Test generation is only one way of *proving existential reachability properties* of programs, where specific concrete input values are generated to exercise specific program paths. More generally, such properties can be proved using so-called *must abstractions* of programs [65], without necessarily generating concrete tests. A must abstraction is defined as a program abstraction that preserves existential reachability properties of the program. Sound path constraints are particular cases of must abstractions [75]. Must abstractions can also be built backwards from error states using static program analysis [33, 84]. This approach can detect program locations and states provably leading to error states (no false alarms), but may fail to prove reachability of those error states from whole-program initial states, and hence may miss bugs or report unreachable error states.

Other Verification Approaches from Programs to Logic. As mentioned earlier in Sect. 19.3.1, test generation using symbolic execution, path constraints and constraint solving is related to other approaches to program verification which reason about programs using logic. Examples of such approaches are verification-condition generation [5, 48], symbolic model checking [18] and SAT/SMT-based bounded model checking [37, 40]. All these approaches have a lot in common, yet differ in important details. In a nutshell, these approaches translate an entire program into a single logic formula using static program analysis. This logic encoding usually tracks both control and data dependencies on all program variables. Program verification is then usually reduced to a validity check using an automated theorem prover. When applicable, these approaches can efficiently prove complex properties of programs. In contrast, test generation using symbolic execution builds a logic representation of a program *incrementally*, one path at a time. Path-by-path program exploration obviously suffers from the path explosion problem discussed earlier, but it scales to large complex programs which are currently beyond the scope of applicability of other automatic program verification approaches like SAT/SMT-based bounded model checking. Verification-condition generation has been shown

to scale to some large programs but it is not automatic: it typically requires a large quantity of nontrivial user annotations, such as loop invariants and function pre/post-conditions, to work in practice and is more similar to interactive theorem proving. We refer the reader to [69] for a more detailed comparison of all these approaches.

19.6 Conclusion

We discussed how model checking can be combined with testing to define a dynamic form of software model checking based on systematic testing, which scales to industrial-size concurrent and data-driven software. This line of work was developed over the last two decades and is still an active area of research. This approach has been implemented in tens of tools by now. The application of those tools has, collectively, found thousands of new bugs, many of them critical from a reliability or security point of view, in many different application domains.

Yet much is still to be accomplished. Software model checking has been successfully applied to several niche applications, such as communication software, device drivers and file parsers, but has remained elusive for general-purpose software. Most tools mentioned in the previous sections are research prototypes, aimed at exploring new ideas, but they are not used on a regular basis by ordinary software developers and testers. Finding other "killer apps" for these techniques, beyond device drivers [4] and file parsers [72], is *critical* in order to sustain progress in this research area.

References

1. Anand, S., Godefroid, P., Tillmann, N.: Demand-driven compositional symbolic execution. In: Ramakrishnan, C.R., Rehof, J. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 4963, pp. 367–381. Springer, Heidelberg (2008)
2. Anand, S., Păsăreanu, C.S., Visser, W.: JPF-SE: a symbolic execution extension to Java PathFinder. In: Grumberg, O., Huth, M. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 4424, pp. 134–138. Springer, Heidelberg (2007)
3. Artzi, S., Kiezun, A., Dolby, J., Tip, F., Dig, D., Paradkar, A.M., Ernst, M.D.: Finding bugs in web applications using dynamic test generation and explicit-state model checking. IEEE Trans. Softw. Eng. **36**(4), 474–494 (2010)
4. Ball, T., Rajamani, S.K.: The SLAM toolkit. In: Berry, G., Comon, H., Finkel, A. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 2102, pp. 260–264. Springer, Heidelberg (2001)
5. Barnett, M., Chang, B.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: a modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) Formal Methods for Components and Objects (FMCO). LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2005)
6. Beckman, N.E., Nori, A.V., Rajamani, S.K., Simmons, R.J.: Proofs from tests. In: Ryder, B.G., Zeller, A. (eds.) Intl. Symp. on Software Testing and Analysis (ISSTA), pp. 3–14. ACM, New York (2008)

7. Bensalem, S., Peled, D., Qu, H., Tripakis, S.: Generating path conditions for timed systems. In: Romijn, J., Smith, G., van de Pol, J. (eds.) Integrated Formal Methods (IFM). LNCS, vol. 3771, pp. 5–19. Springer, Heidelberg (2005)
8. Bensalem, S., Peled, D., Qu, H., Tripakis, S., Zuck, L.D.: Test case generation for ultimately periodic paths. In: Yorav, K. (ed.) Intl. Haifa Verification Conf. (HVC). LNCS, vol. 4899, pp. 120–135. Springer, Heidelberg (2007)
9. Beyer, D., Chlipala, A., Henzinger, T.A., Jhala, R., Majumdar, R.: Generating tests from counterexamples. In: Finkelstein, A., Estublier, J., Rosenblum, D.S. (eds.) Intl. Conf. on Software Engineering (ICSE), pp. 326–335. IEEE, Piscataway (2004)
10. Beyer, D., Henzinger, T.A., Keremoglu, M.E., Wendler, P.: Conditional model checking: a technique to pass information between verifiers. In: Tracz, W., Robillard, M.P., Bultan, T. (eds.) Intl. Symp. on Foundations of Software Engineering (FSE), p. 57. ACM, New York (2012)
11. Beyer, D., Henzinger, T.A., Théoduloz, G.: Program analysis with dynamic precision adjustment. In: Intl. Conf. on Automated Software Engineering (ASE), pp. 29–38. IEEE, Piscataway (2008)
12. Beyer, D., Holzer, A., Tautschnig, M., Veith, H.: Information reuse for multi-goal reachability analyses. In: Felleisen, M., Gardner, P. (eds.) European Symp. on Programming (ESOP). LNCS, vol. 7792, pp. 472–491. Springer, Heidelberg (2013)
13. van der Bijl, M., Rensink, A., Tretmans, J.: Compositional testing with ioco. In: Intl. Workshop on Formal Approaches to Testing of Software (FATES). LNCS, vol. 2931, pp. 86–100. Springer, Heidelberg (2003)
14. Boonstoppel, P., Cadar, C., Engler, D.R.: Rwset: attacking path explosion in constraint-based test generation. In: Ramakrishnan, C.R., Rehof, J. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 4963, pp. 351–366. Springer, Heidelberg (2008)
15. Bounimova, E., Godefroid, P., Molnar, D.A.: Billions and billions of constraints: whitebox fuzz testing in production. In: Notkin, D., Cheng, B.H.C., Pohl, K. (eds.) Intl. Conf. on Software Engineering (ICSE), pp. 122–131. IEEE/ACM, Piscataway/New York (2013)
16. Boyer, R.S., Elspas, B., Levitt, K.N.: SELECT—a formal system for testing and debugging programs by symbolic execution. ACM SIGPLAN Not. **10**(6), 234–245 (1975)
17. Brat, G.P., Drusinsky, D., Giannakopoulou, D., Goldberg, A., Havelund, K., Lowry, M.R., Păsăreanu, C.S., Venet, A., Visser, W., Washington, R.: Experimental evaluation of verification and validation tools on Martian Rover software. Form. Methods Syst. Des. **25**(2–3), 167–198 (2004)
18. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. In: Symp. on Logic in Computer Science (LICS), pp. 428–439. IEEE, Piscataway (1990)
19. Burckhardt, S., Kothari, P., Musuvathi, M., Nagarakatte, S.: A randomized scheduler with probabilistic guarantees of finding bugs. In: Hoe, J.C., Adve, V.S. (eds.) Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 167–178. ACM, New York (2010)
20. Burnim, J., Jalbert, N., Stergiou, C., Sen, K.: Looper: lightweight detection of infinite loops at runtime. In: Intl. Conf. on Automated Software Engineering (ASE), pp. 161–169. IEEE, Piscataway (2009)
21. Burnim, J., Juvekar, S., Sen, K.: WISE: automated test generation for worst-case complexity. In: Intl. Conf. on Software Engineering (ICSE), pp. 463–473. IEEE, Piscataway (2009)
22. Burnim, J., Necula, G., Sen, K.: Separating functional and parallel correctness using nondeterministic sequential specifications. In: USENIX Workshop on Hot Topics in Parallelism (HotPar). USENIX Association, Berkeley (2010)
23. Burnim, J., Necula, G.C., Sen, K.: Specifying and checking semantic atomicity for multithreaded programs. In: Gupta, R., Mowry, T.C. (eds.) Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 79–90. ACM, New York (2011)

24. Burnim, J., Sen, K.: Heuristics for scalable dynamic test generation. In: Intl. Conf. on Automated Software Engineering (ASE), pp. 443–446. IEEE, Piscataway (2008)
25. Burnim, J., Sen, K.: Asserting and checking determinism for multithreaded programs. Commun. ACM **53**(6), 97–105 (2010)
26. Burnim, J., Sen, K., Stergiou, C.: Sound and complete monitoring of sequential consistency for relaxed memory models. In: Abdulla, P.A., Leino, K.R.M. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 6605, pp. 11–25. Springer, Heidelberg (2011)
27. Burnim, J., Sen, K., Stergiou, C.: Testing concurrent programs on relaxed memory models. In: Dwyer, M.B., Tip, F. (eds.) Intl. Symp. on Software Testing and Analysis (ISSTA), pp. 122–132. ACM, New York (2011)
28. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Draves, R., van Renesse, R. (eds.) Operating Systems Design and Implementation (OSDI), pp. 209–224. USENIX Association, Berkeley (2008)
29. Cadar, C., Engler, D.R.: Execution generated test cases: how to make systems code crash itself. In: Godefroid, P. (ed.) Intl. Symp. on Model Checking of Software (SPIN). LNCS, vol. 3639, pp. 2–23. Springer, Heidelberg (2005)
30. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: EXE: automatically generating inputs of death. In: Juels, A., Wright, R.N., di Vimercati, S.D.C. (eds.) ACM Conf. on Computer and Communications Security (CCS), pp. 322–335. ACM, New York (2006)
31. Cadar, C., Godefroid, P., Khurshid, S., Păsăreanu, C.S., Sen, K., Tillmann, N., Visser, W.: Symbolic execution for software testing in practice: preliminary assessment. In: Taylor, R.N., Gall, H.C., Medvidovic, N. (eds.) Intl. Conf. on Software Engineering (ICSE), pp. 1066–1071. ACM, New York (2011)
32. Cadar, C., Sen, K.: Symbolic execution for software testing: three decades later. Commun. ACM **56**(2), 82–90 (2013)
33. Chandra, S., Fink, S.J., Sridharan, M.: Snugglebug: a powerful approach to weakest preconditions. In: Hind, M., Diwan, A. (eds.) Conf. on Programming Language Design and Implementation (PLDI), pp. 363–374. ACM, New York (2009)
34. Chandra, S., Godefroid, P., Palm, C.: Software model checking in practice: an industrial case study. In: Tracz, W., Young, M., Magee, J. (eds.) Intl. Conf. on Software Engineering (ICSE), pp. 431–441. ACM, New York (2002)
35. Chang, J., Richardson, D.J., Sankar, S.: Structural specification-based testing with ADL. In: Intl. Symp. on Software Testing and Analysis (ISSTA), pp. 62–70. ACM, New York (1996)
36. Chipounov, V., Kuznetsov, V., Candea, G.: S2E: a platform for in-vivo multi-path analysis of software systems. In: Gupta, R., Mowry, T.C. (eds.) Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 265–278. ACM, New York (2011)
37. Clarke, E.M., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Form. Methods Syst. Des. **19**(1), 7–34 (2001)
38. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching-time temporal logic. In: Kozen, D. (ed.) Workshop on Logics of Programs. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1981)
39. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Trans. Program. Lang. Syst. **8**(2), 244–263 (1986)
40. Clarke, E.M., Kroening, D., Yorav, K.: Behavioral consistency of C and Verilog programs using bounded model checking. In: Design Automation Conf. (DAC), pp. 368–371. ACM, New York (2003)

41. Clarke, L.A.: A program testing system. In: ACM, vol. 176, pp. 488–491 (1976)
42. Clarke, L.A., Richardson, D.J.: Applications of symbolic evaluation. J. Syst. Softw. **5**(1), 15–35 (1985)
43. Colby, C.: Analyzing the communication topology of concurrent programs. In: Jones, N.D. (ed.) Symposium on Partial Evaluation and Semantics-Based Program Manipulation, pp. 202–213. ACM, New York (1995)
44. Corbett, J.C.: Constructing abstract models of concurrent real-time software. In: Intl. Symp. on Software Testing and Analysis (ISSTA), pp. 250–260 (1996)
45. Corbett, J.C., Dwyer, M.B., Hatcliff, J., Laubach, S., Păsăreanu, C.S., Robby, Zheng, H.: Bandera: extracting finite-state models from Java source code. In: Ghezzi, C., Jazayeri, M., Wolf, A.L. (eds.) Intl. Conf. on Software Engineering (ICSE), pp. 439–448. ACM, New York (2000)
46. Cridlig, R.: Semantic analysis of shared-memory concurrent languages using abstract model-checking. In: Jones, N.D. (ed.) Symposium on Partial Evaluation and Semantics-Based Program Manipulation, pp. 214–225. ACM, New York (1995)
47. Csallner, C., Smaragdakis, Y.: Check'n'crash: combining static checking and testing. In: Roman, G., Griswold, W.G., Nuseibeh, B. (eds.) Intl. Conf. on Software Engineering (ICSE), pp. 422–431. ACM, New York (2005)
48. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM **18**(8), 453–457 (1975)
49. Dillon, L.K., Yu, Q.: Oracles for checking temporal properties of concurrent systems. In: Wile, D.S. (ed.) Intl. Symp. on Foundations of Software Engineering (FSE), pp. 140–153. ACM, New York (1994)
50. Drusinsky, D.: The temporal rover and the ATG rover. In: Havelund, K., Penix, J., Visser, W. (eds.) Intl. Symp. on Model Checking of Software (SPIN). LNCS, vol. 1885, pp. 323–330. Springer, Heidelberg (2000)
51. Edelstein, O., Farchi, E., Goldin, E., Nir, Y., Ratsaby, G., Ur, S.: Framework for testing multi-threaded Java programs. Concurr. Comput. **15**(3–5), 485–499 (2003)
52. Elkarablieh, B., Godefroid, P., Levin, M.Y.: Precise pointer reasoning for dynamic test generation. In: Rothermel, G., Dillon, L.K. (eds.) Intl. Symp. on Software Testing and Analysis (ISSTA), pp. 129–140. ACM, New York (2009)
53. Emmi, M., Majumdar, R., Sen, K.: Dynamic test input generation for database applications. In: Rosenblum, D.S., Elbaum, S.G. (eds.) Intl. Symp. on Software Testing and Analysis (ISSTA), pp. 151–162. ACM, New York (2007)
54. Ernst, M.D.: Static and dynamic analysis: synergy and duality. In: ICSE Workshop on Dynamic Analysis (WODA), pp. 25–28. ACM, New York (2003)
55. Farzan, A., Holzer, A., Razavi, N., Veith, H.: Con2colic testing. In: Meyer, B., Baresi, L., Mezini, M. (eds.) Intl. Symp. on Foundations of Software Engineering (FSE), pp. 37–47. ACM, New York (2013)
56. Fernandez, J., Jard, C., Jéron, T., Viho, C.: Using on-the-fly verification techniques for the generation of test suites. In: Alur, R., Henzinger, T.A. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 1102, pp. 348–359. Springer, Heidelberg (1996)
57. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: Palsberg, J., Abadi, M. (eds.) Symp. on Principles of Programming Languages (POPL), pp. 110–121. ACM, New York (2005)
58. Floyd, R.W.: Assigning meanings to programs. In: Mathematical Aspects of Computer Science, vol. 19, pp. 19–32 (1967)
59. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems—An Approach to the State-Explosion Problem. LNCS, vol. 1032. Springer, Heidelberg (1996)
60. Godefroid, P.: Model checking for programming languages using VeriSoft. In: Lee, P., Henglein, F., Jones, N.D. (eds.) Symp. on Principles of Programming Languages (POPL), pp. 174–186. ACM, New York (1997)
61. Godefroid, P.: Software model checking: the VeriSoft approach. Form. Methods Syst. Des. **26**(2), 77–101 (2005)

62. Godefroid, P.: Compositional dynamic test generation. In: Hofmann, M., Felleisen, M. (eds.) Symp. on Principles of Programming Languages (POPL), pp. 47–54. ACM, New York (2007)
63. Godefroid, P.: Higher-order test generation. In: Hall, M.W., Padua, D.A. (eds.) Conf. on Programming Language Design and Implementation (PLDI), pp. 258–269. ACM, New York (2011)
64. Godefroid, P., Hanmer, R.S., Jagadeesan, L.J.: Model checking without a model: an analysis of the heart-beat monitor of a telephone switch using VeriSoft. In: Intl. Symp. on Software Testing and Analysis (ISSTA), pp. 124–133 (1998)
65. Godefroid, P., Huth, M., Jagadeesan, R.: Abstraction-based model checking using modal transition systems. In: Larsen, K.G., Nielsen, M. (eds.) Intl. Conf. on Concurrency Theory (CONCUR). LNCS, vol. 2154, pp. 426–440. Springer, Heidelberg (2001)
66. Godefroid, P., Kiezun, A., Levin, M.Y.: Grammar-based whitebox fuzzing. In: Conf. on Programming Language Design and Implementation (PLDI), pp. 206–215. ACM, New York (2008)
67. Godefroid, P., Kinder, J.: Proving memory safety of floating-point computations by combining static and dynamic program analysis. In: Tonella, P., Orso, A. (eds.) Intl. Symp. on Software Testing and Analysis (ISSTA), pp. 1–12. ACM, New York (2010)
68. Godefroid, P., Klarlund, N., Sen, K.: DART: Directed Automated Random Testing. In: Sarkar, V., Hall, M.W. (eds.) Conf. on Programming Language Design and Implementation (PLDI), pp. 213–223. ACM, New York (2005)
69. Godefroid, P., Lahiri, S.K.: From program to logic: an introduction. In: Meyer, B., Nordio, M. (eds.) Tools for Practical Software Verification (LASER). LNCS, vol. 7682, pp. 31–44. Springer, Heidelberg (2012)
70. Godefroid, P., Lahiri, S.K., Rubio-González, C.: Statically validating must summaries for incremental compositional dynamic test generation. In: Yahav, E. (ed.) Intl. Symp. on Static Analysis (SAS). LNCS, vol. 6887, pp. 112–128. Springer, Heidelberg (2011)
71. Godefroid, P., Levin, M.Y., Molnar, D.A.: Active property checking. In: de Alfaro, L., Palsberg, J. (eds.) Intl. Conf. on Embedded Software (EMSOFT), pp. 207–216. ACM, New York (2008)
72. Godefroid, P., Levin, M.Y., Molnar, D.A.: Automated whitebox fuzz testing. In: Network and Distributed System Security Symposium (NDSS). The Internet Society, Reston (2008)
73. Godefroid, P., Levin, M.Y., Molnar, D.A.: SAGE: whitebox fuzzing for security testing. Commun. ACM **55**(3), 40–44 (2012)
74. Godefroid, P., Luchaup, D.: Automatic partial loop summarization in dynamic test generation. In: Dwyer, M.B., Tip, F. (eds.) Intl. Symp. on Software Testing and Analysis (ISSTA), pp. 23–33. ACM, New York (2011)
75. Godefroid, P., Nori, A.V., Rajamani, S.K., Tetali, S.: Compositional may-must program analysis: unleashing the power of alternation. In: Hermenegildo, M.V., Palsberg, J. (eds.) Symp. on Principles of Programming Languages (POPL), pp. 43–56. ACM, New York (2010)
76. Godefroid, P., Taly, A.: Automated synthesis of symbolic instruction encodings from I/O samples. In: Vitek, J., Lin, H., Tip, F. (eds.) Conf. on Programming Language Design and Implementation (PLDI), pp. 441–452. ACM, New York (2012)
77. Gulavani, B.S., Henzinger, T.A., Kannan, Y., Nori, A.V., Rajamani, S.K.: SYNERGY: a new algorithm for property checking. In: Young, M., Devanbu, P.T. (eds.) Intl. Symp. on Foundations of Software Engineering (FSE), pp. 117–127. ACM, New York (2006)
78. Gunter, E.L., Peled, D.: Path exploration tool. In: Cleaveland, R. (ed.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 1579, pp. 405–419. Springer, Heidelberg (1999)
79. Gunter, E.L., Peled, D.: Model checking, testing and verification working together. Form. Asp. Comput. **17**(2), 201–221 (2005)
80. Gupta, N., Mathur, A.P., Soffa, M.L.: Generating test data for branch coverage. In: Intl. Conf. on Automated Software Engineering (ASE), pp. 219–228. IEEE, Piscataway (2000)

81. Havelund, K., Rosu, G.: Monitoring Java programs with Java PathExplorer. Electron. Notes Theor. Comput. Sci. **55**(2), 200–217 (2001)
82. Helmstetter, C., Maraninchi, F., Maillet-Contoz, L., Moy, M.: Automatic generation of schedulings for improving the test coverage of systems-on-a-chip. In: Formal Methods in Computer Aided Design (FMCAD), pp. 171–178. IEEE, Piscataway (2006)
83. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
84. Hoenicke, J., Leino, K.R.M., Podelski, A., Schäf, M., Wies, T.: It's doomed; we can prove it. In: Cavalcanti, A., Dams, D. (eds.) World Congress on Formal Methods. LNCS, vol. 5850, pp. 338–353. Springer, Heidelberg (2009)
85. Holzmann, G.J., Smith, M.H.: A practical method for verifying event-driven software. In: Boehm, B.W., Garlan, D., Kramer, J. (eds.) Intl. Conf. on Software Engineering (ICSE), pp. 597–607. ACM, New York (1999)
86. Howden, W.E.: Symbolic testing and the DISSECT symbolic evaluation system. IEEE Trans. Softw. Eng. **3**(4), 266–278 (1977)
87. Jagadeesan, L.J., Porter, A.A., Puchol, C., Ramming, J.C., Votta, L.G.: Specification-based testing of reactive software: tools and experiments (experience report). In: Adrion, W.R., Fuggetta, A., Taylor, R.N., Wasserman, A.I. (eds.) Intl. Conf. on Software Engineering (ICSE), pp. 525–535. ACM, New York (1997)
88. Jalbert, N., Sen, K.: A trace simplification technique for effective debugging of concurrent programs. In: Roman, G., Sullivan, K.J. (eds.) Intl. Symp. on Foundations of Software Engineering (FSE), pp. 57–66. ACM, New York (2010)
89. Joshi, P., Naik, M., Park, C., Sen, K.: CalFuzzer: an extensible active testing framework for concurrent programs. In: Bouajjani, A., Maler, O. (eds.) Intl. Conf. on Computer-Aided Verification (CAV), pp. 675–681 (2009)
90. Joshi, P., Naik, M., Sen, K., Gay, D.: An effective dynamic analysis for detecting generalized deadlocks. In: Roman, G., Sullivan, K.J. (eds.) Intl. Symp. on Foundations of Software Engineering (FSE), pp. 327–336. ACM, New York (2010)
91. Joshi, P., Park, C., Sen, K., Naik, M.: A randomized dynamic program analysis technique for detecting real deadlocks. In: Hind, M., Diwan, A. (eds.) Conf. on Programming Language Design and Implementation (PLDI), pp. 110–120. ACM, New York (2009)
92. Joshi, P., Sen, K., Shlimovich, M.: Predictive testing: amplifying the effectiveness of software testing. In: Crnkovic, I., Bertolino, A. (eds.) Intl. Symp. on Foundations of Software Engineering (FSE), pp. 561–564 (2007)
93. Kannan, Y., Sen, K.: Universal symbolic execution and its application to likely data structure invariant generation. In: Ryder, B.G., Zeller, A. (eds.) Intl. Symp. on Software Testing and Analysis (ISSTA), pp. 283–294. ACM, New York (2008)
94. Khurshid, S., Păsăreanu, C.S., Visser, W.: Generalized symbolic execution for model checking and testing. In: Garavel, H., Hatcliff, J. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 2619, pp. 553–568. Springer, Heidelberg (2003)
95. Killian, C.E., Anderson, J.W., Jhala, R., Vahdat, A.: Life, death, and the critical transition: finding liveness bugs in systems code. In: Balakrishnan, H., Druschel, P. (eds.) USENIX Symp. on Networked Systems Design and Implementation (NSDI). USENIX Association, Berkeley (2007)
96. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976)
97. Korel, B.: A dynamic approach of test data generation. In: IEEE Conference on Software Maintenance, pp. 311–317. IEEE, Piscataway (1990)
98. Kuznetsov, V., Chipounov, V., Candea, G.: Testing closed-source binary device drivers with DDT. In: Barham, P., Roscoe, T. (eds.) USENIX Annual Technical Conference. USENIX Association, Berkeley (2010)
99. Lichtenstein, O., Pnueli, A.: Checking that finite state concurrent programs satisfy their linear specification. In: Deusen, M.S.V., Galil, Z., Reid, B.K. (eds.) Symp. on Principles of Programming Languages (POPL), pp. 97–107. ACM, New York (1985)

100. Long, D.L., Clarke, L.A.: Data flow analysis of concurrent systems that use the rendezvous model of synchronization. In: Symposium on Testing, Analysis, and Verification, pp. 21–35 (1991)
101. Majumdar, R., Sen, K.: Hybrid concolic testing. In: Intl. Conf. on Software Engineering (ICSE), pp. 416–426. IEEE, Piscataway (2007)
102. Majumdar, R., Xu, R.: Directed test generation using symbolic grammars. In: Stirewalt, R.E.K., Egyed, A., Fischer, B. (eds.) Intl. Conf. on Automated Software Engineering (ASE), pp. 134–143. ACM, New York (2007)
103. Majumdar, R., Xu, R.: Reducing test inputs using information partitions. In: Bouajjani, A., Maler, O. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 5643, pp. 555–569. Springer, Heidelberg (2009)
104. Masticola, S.P., Ryder, B.G.: Non-concurrency analysis. In: Chen, M.C., Halstead, R. (eds.) Symposium on Principles and Practice of Parallel Programming (PPOPP), pp. 129–138. ACM, New York (1993)
105. Molnar, D.A., Wagner, D.: CatchConv: symbolic execution and run-time type inference for integer conversion errors. Tech. Rep. UCB/EECS-2007-23, EECS Department, University of California, Berkeley (2007)
106. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
107. Musuvathi, M., Engler, D.R.: Model checking large network protocol implementations. In: Morris, R., Savage, S. (eds.) USENIX Symp. on Networked Systems Design and Implementation (NSDI), pp. 155–168. USENIX Association, Berkeley (2004)
108. Musuvathi, M., Park, D.Y.W., Chou, A., Engler, D.R., Dill, D.L.: CMC: a pragmatic approach to model checking real code. In: Culler, D.E., Druschel, P. (eds.) Operating Systems Design and Implementation (OSDI), pp. 75–88. USENIX Association, Berkeley (2002)
109. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In: Ferrante, J., McKinley, K.S. (eds.) Conf. on Programming Language Design and Implementation (PLDI), pp. 446–455. ACM, New York (2007)
110. Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P.A., Neamtiu, I.: Finding and reproducing heisenbugs in concurrent programs. In: Draves, R., van Renesse, R. (eds.) Operating Systems Design and Implementation (OSDI), pp. 267–280. USENIX Association, Berkeley (2008)
111. Namjoshi, K.S., Kurshan, R.P.: Syntactic program transformations for automatic abstraction. In: Emerson, E.A., Sistla, A.P. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 1855, pp. 435–449. Springer, Heidelberg (2000)
112. Necula, G.C., McPeak, S., Weimer, W.: CCured: type-safe retrofitting of legacy code. In: Launchbury, J., Mitchell, J.C. (eds.) Symp. on Principles of Programming Languages (POPL), pp. 128–139. ACM, New York (2002)
113. Nori, A.V., Rajamani, S.K.: An empirical study of optimizations in YOGI. In: Kramer, J., Bishop, J., Devanbu, P.T., Uchitel, S. (eds.) Intl. Conf. on Software Engineering (ICSE), pp. 355–364. ACM, New York (2010)
114. Offutt, A.J., Jin, Z., Pan, J.: The dynamic domain reduction procedure for test data generation. Softw. Pract. Exp. **29**(2), 167–193 (1999)
115. Park, C., Sen, K.: Randomized active atomicity violation detection in concurrent programs. In: Harrold, M.J., Murphy, G.C. (eds.) Intl. Symp. on Foundations of Software Engineering (FSE), pp. 135–145. ACM, New York (2008)
116. Park, C., Sen, K., Hargrove, P., Iancu, C.: Efficient data race detection for distributed memory parallel programs. In: Lathrop, S., Costa, J., Kramer, W. (eds.) Conference on High Performance Computing Networking, Storage and Analysis (SC), pp. 51:1–51:12. ACM, New York (2011)
117. Park, C., Sen, K., Iancu, C.: Scaling data race detection for partitioned global address space programs. In: Malony, A.D., Nemirovsky, M., Midkiff, S.P. (eds.) International Conference on Supercomputing (ICS), pp. 47–58. ACM, New York (2013)

118. Peled, D.: All from one, one for all: on model checking using representatives. In: Courcoubetis, C. (ed.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 697, pp. 409–423. Springer, Heidelberg (1993)
119. Penix, J., Visser, W., Park, S., Păsăreanu, C.S., Engstrom, E., Larson, A., Weininger, N.: Verifying time partitioning in the DEOS scheduling kernel. Form. Methods Syst. Des. **26**(2), 103–135 (2005)
120. Person, S., Dwyer, M.B., Elbaum, S.G., Păsăreanu, C.S.: Differential symbolic execution. In: Harrold, M.J., Murphy, G.C. (eds.) Intl. Symp. on Foundations of Software Engineering (FSE), pp. 226–237. ACM, New York (2008)
121. Person, S., Yang, G., Rungta, N., Khurshid, S.: Directed incremental symbolic execution. In: Hall, M.W., Padua, D.A. (eds.) Conf. on Programming Language Design and Implementation (PLDI), pp. 504–515. ACM, New York (2011)
122. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: Halbwachs, N., Zuck, L.D. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 3440, pp. 93–107. Springer, Heidelberg (2005)
123. Qadeer, S., Wu, D.: KISS: keep it simple and sequential. In: Pugh, W., Chambers, C. (eds.) Conf. on Programming Language Design and Implementation (PLDI), pp. 14–24. ACM, New York (2004)
124. Queille, J., Sifakis, J.: Specification and verification of concurrent systems in CESAR. In: Dezani-Ciancaglini, M., Montanari, U. (eds.) International Symposium on Programming. LNCS, vol. 137, pp. 337–351. Springer, Heidelberg (1982)
125. Ramamoorthy, C.V., Ho, S.F., Chen, W.T.: On the automated generation of program test data. IEEE Trans. Softw. Eng. **2**(4), 293–300 (1976)
126. Razavi, N., Ivancic, F., Kahlon, V., Gupta, A.: Concurrent test generation using concolic multi-trace analysis. In: Jhala, R., Igarashi, A. (eds.) Asian Symp. on Programming Languages and Systems (APLAS), pp. 239–255. Springer, Heidelberg (2012)
127. Richardson, D.J.: TAOS: testing with analysis and oracle support. In: Intl. Symp. on Software Testing and Analysis (ISSTA), pp. 138–153 (1994)
128. Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., Song, D.: A symbolic execution framework for JavaScript. In: Symposium on Security and Privacy, pp. 513–528. IEEE, Piscataway (2010)
129. Saxena, P., Poosankam, P., McCamant, S., Song, D.: Loop-extended symbolic execution on binary programs. In: Rothermel, G., Dillon, L.K. (eds.) Intl. Symp. on Software Testing and Analysis (ISSTA), pp. 225–236. ACM, New York (2009)
130. Sen, K.: CATG: a concolic testing tool for sequential Java programs. https://github.com/ksen007/janala2
131. Sen, K.: Scalable automated methods for dynamic program analysis. Ph.D. thesis, University of Illinois at Urbana-Champaign (2006)
132. Sen, K.: Effective random testing of concurrent programs. In: Stirewalt, R.E.K., Egyed, A., Fischer, B. (eds.) Intl. Conf. on Automated Software Engineering (ASE), pp. 323–332. ACM, New York (2007)
133. Sen, K.: Race directed random testing of concurrent programs. In: Gupta, R., Amarasinghe, S.P. (eds.) Conf. on Programming Language Design and Implementation (PLDI), pp. 11–21. ACM, New York (2008)
134. Sen, K., Agha, G.: Automated systematic testing of open distributed programs. In: Baresi, L., Heckel, R. (eds.) Intl. Conf. on Fundamental Approaches to Software Engineering (FASE). LNCS, vol. 3922, pp. 339–356. Springer, Heidelberg (2006)
135. Sen, K., Agha, G.: CUTE and jCUTE: concolic unit testing and explicit path model-checking tools. In: Ball, T., Jones, R.B. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 4144, pp. 419–423. Springer, Heidelberg (2006)
136. Sen, K., Agha, G.: A race-detection and flipping algorithm for automated testing of multithreaded programs. In: Bin, E., Ziv, A., Ur, S. (eds.) Intl. Haifa Verification Conf. (HVC). LNCS, vol. 4383, pp. 166–182. Springer, Heidelberg (2006)

137. Sen, K., Kalasapur, S., Brutch, T.G., Gibbs, S.: Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. In: Meyer, B., Baresi, L., Mezini, M. (eds.) Intl. Symp. on Foundations of Software Engineering (FSE), pp. 488–498. ACM, New York (2013)
138. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: Wermelinger, M., Gall, H.C. (eds.) Intl. Symp. on Foundations of Software Engineering (FSE), pp. 263–272. ACM, New York (2005)
139. Siegel, S.F., Mironova, A., Avrunin, G.S., Clarke, L.A.: Using model checking with symbolic execution to verify parallel numerical programs. In: Pollock, L.L., Pezzè, M. (eds.) Intl. Symp. on Software Testing and Analysis (ISSTA), pp. 157–168. ACM, New York (2006)
140. Song, D.X., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M.G., Liang, Z., Newsome, J., Poosankam, P., Saxena, P.: BitBlaze: a new approach to computer security via binary analysis. In: Sekar, R., Pujari, A.K. (eds.) International Conference on Information Systems Security (ICISS). LNCS, vol. 5352, pp. 1–25. Springer, Heidelberg (2008)
141. Stoller, S.D.: Model-checking multi-threaded distributed java programs. In: Havelund, K., Penix, J., Visser, W. (eds.) Intl. Symp. on Model Checking of Software (SPIN). LNCS, vol. 1885. Springer, Heidelberg (2000)
142. Taylor, R.N.: A general-purpose algorithm for analyzing concurrent programs. Commun. ACM **26**(5), 362–376 (1983)
143. Tillmann, N., de Halleux, J.: Pex—white box test generation for net. In: Beckert, B., Hähnle, R. (eds.) Intl. Conf. on Tests and Proofs (TAP). LNCS, vol. 4966, pp. 134–153. Springer, Heidelberg (2008)
144. Valmari, A.: Stubborn sets for reduced state space generation. In: Rozenberg, G. (ed.) International Conference on Applications and Theory of Petri Nets. LNCS, vol. 483, pp. 491–515. Springer, Heidelberg (1989)
145. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification. In: Symp. on Logic in Computer Science (LICS), pp. 332–344. IEEE, Piscataway (1986)
146. Veanes, M., Campbell, C., Grieskamp, W., Schulte, W., Tillmann, N., Nachmanson, L.: Model-based testing of object-oriented reactive systems with Spec Explorer. In: Hierons, R.M., Bowen, J.P., Harman, M. (eds.) Formal Methods and Testing. LNCS, vol. 4949, pp. 39–76. Springer, Heidelberg (2008)
147. Venet, A.: Abstract interpretation of the pi-calculus. In: Dam, M. (ed.) Analysis and Verification of Multiple-Agent Languages (LOMAPS). LNCS, vol. 1192, pp. 51–75. Springer, Heidelberg (1996)
148. Visser, W., Havelund, K., Brat, G.P., Park, S.: Model checking programs. In: Intl. Conf. on Automated Software Engineering (ASE), pp. 3–12. IEEE, Piscataway (2000)
149. Visser, W., Păsăreanu, C.S., Khurshid, S.: Test input generation with Java PathFinder. In: Avrunin, G.S., Rothermel, G. (eds.) Intl. Symp. on Software Testing and Analysis (ISSTA), pp. 97–107. ACM, New York (2004)
150. Vo, A., Vakkalanka, S.S., Delisi, M., Gopalakrishnan, G., Kirby, R.M., Thakur, R.: Formal verification of practical MPI programs. In: Reed, D.A., Sarkar, V. (eds.) Symposium on Principles and Practice of Parallel Programming (PPOPP), pp. 261–270. ACM, New York (2009)
151. Williams, N., Marre, B., Mouy, P., Roger, M.: Pathcrawler: automatic generation of path tests by combining static and dynamic analysis. In: Cin, M.D., Kaâniche, M., Pataricza, A. (eds.) European Dependable Computing Conference (EDCC), pp. 281–292. Springer, Heidelberg (2005)
152. Xu, R., Godefroid, P., Majumdar, R.: Testing for buffer overflows with length abstraction. In: Ryder, B.G., Zeller, A. (eds.) Intl. Symp. on Software Testing and Analysis (ISSTA), pp. 27–38. ACM, New York (2008)
153. Yang, J., Chen, T., Wu, M., Xu, Z., Liu, X., Lin, H., Yang, M., Long, F., Zhang, L., Zhou, L.: MODIST: transparent model checking of unmodified distributed systems. In: Rexford, J., Sirer, E.G. (eds.) USENIX Symp. on Networked Systems Design and Implementation (NSDI), pp. 213–228. USENIX Association, Berkeley (2009)

154. Yang, J., Twohey, P., Engler, D.R., Musuvathi, M.: Using model checking to find serious file system errors. In: Brewer, E.A., Chen, P. (eds.) Operating Systems Design and Implementation (OSDI), pp. 273–288. USENIX Association, Berkeley (2004)
155. Yannakakis, M., Lee, D.: Testing finite state machines (extended abstract). In: Koutsougeras, C., Vitter, J.S. (eds.) ACM Symp. on Theory of Computing (STOC), pp. 476–485. ACM, New York (1991)

Chapter 20
Combining Model Checking and Deduction

Natarajan Shankar

Abstract There are two basic approaches to automated verification. In model checking, the system is viewed as a graph representing possible execution steps. Properties are established by exploring or traversing the graph structure. In deduction, both the system and its putative properties are represented by formulas in a logic, and the resulting proof obligations are discharged by decision procedures or by automated or semi-automated proof construction. Model checking sacrifices expressivity for greater automation, and with deduction it is vice versa. Newer techniques combine deductive and model-checking approaches to achieve greater scale, expressivity, and automation. We examine the logical foundations of the two approaches and explore their similarities, differences, and complementarities. The presentation is directed at students and researchers who are interested in understanding the research challenges at the intersection of deduction and model checking.

20.1 Introduction

Verification establishes properties that hold of all the possible executions of a program. There are two basic approaches to verification. In model checking [24], the system is described as a *model* M which is a graph where the vertices are the states of the computation and the edges are possible execution steps. A property is a formula P in a logic that characterizes a class of computations that can be generated from the graph. For example, a mutual exclusion property might state that no two processes are simultaneously in their critical section, meaning that any state violating this property must be unreachable through any valid computation. Another property might assert that any process that is trying to enter its critical section eventually succeeds, meaning that there is no computation from a state where a process is trying to enter its critical section where it never actually enters its critical section. The satisfaction relation $M \models P$ is used to check whether the system M satisfies the property P, i.e., no computation that can be generated from M, violates P.

N. Shankar (✉)
SRI International, Menlo Park, CA, USA
e-mail: shankar@csl.sri.com

E.M. Clarke et al. (eds.), *Handbook of Model Checking*,
DOI 10.1007/978-3-319-10575-8_20

In the deductive approach [18, 78], both the system M and the property P are interpreted as formulas in a logic expressing constraints on the possible executions. The goal is to prove an assertion of the form $M \vdash P$ which captures the judgement that all possible executions of M must satisfy the property P. A property might assert that a binary search procedure for a key k in a sorted array a terminates by returning an index i such that $a[i] = k$ if the element k occurs in the array.

The key distinction between the two approaches is that deductive approaches work on the syntactic structure of the program viewed as a formula to build a proof that $M \vdash P$, whereas model checking works on the graph structure of the computation generated by the program to check that $M \models P$. In terms of automation, model checking verifies $M \models P$ using algorithms that explore the graph structure of M, whereas deduction applies proof rules to the syntactic structure of the formulas M and P. When model checking fails, it produces a concrete counterexample explaining how the model M violates property P. Deductive methods can also produce counterexamples, but typically human insight is required to discriminate between an incorrect system M, an invalid property P, and a misdirected proof attempt. In the latter case, human interaction is needed to redirect proof construction. When deduction succeeds, it yields a proof explaining why property P holds for the system M.

The trend in model checking is toward greater expressivity in systems and properties while preserving the level of automation, while the trend in deduction is toward greater automation. In recent years, there has been a convergence of ideas and techniques in these two approaches to verification. We present the logical foundations of deduction and model checking, and examine their similarities, differences, and complementarities.

Early approaches to verification were based on deductive techniques. Examples of assertional program proofs were given by Goldstine and von Neumann [59, 60] and by Turing [45, 56, 79], but these were not presented as formal proof systems. McCarthy's method [53] for reasoning about recursive Lisp functions uses equational logic and an inference technique called recursion–induction. Floyd's method [34] is applied to programs represented as flowcharts annotated with assertions. A program flowchart (see Fig. 1) can be seen as a directed graph with statements, statement blocks, or decision conditions on the vertices, and assertions on the edges. The flowchart has a source (*start*) vertex with a single outgoing edge, the *precondition*, and a sink (*halt*) vertex with a single incoming edge, the *post-condition*. *Verification conditions* are generated to verify that whenever the assertion for an incoming edge holds and a vertex is executed, the assertion for the corresponding outgoing edge holds. Proving these verification conditions is an effective way of establishing the *partial correctness* of the program: every execution of the program starting in the start vertex, satisfying the precondition, and terminating in the halt vertex, satisfies the post-condition.

Since the flowchart might have loops, it is possible to have valid infinite executions of the flowchart program that do not reach the *halt* vertex. *Total correctness* requires a proof of *termination* showing that every execution starting at the start vertex with a program state satisfying the precondition terminates in the halt vertex.

```
max = 0;
i = 0;
{i ≤ N ∧ ∀(j < i): a[j] ≤ max}
while (i < N){
 if (a[i] > max){
   max = a[i];
   };
 i++;
 }
{∀(j < N): a[j] ≤ max}
```

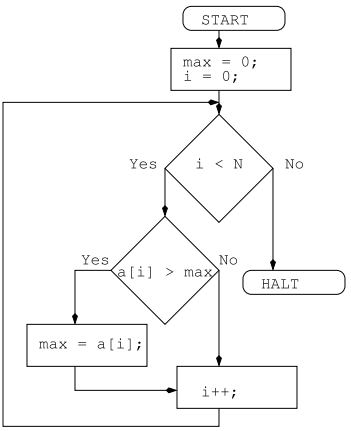


Fig. 1 Program and flowchart for finding the maximal element in a non-negative array

Termination is verified using a *ranking* or *variant* function on the state that associates an ordinal with each state. The verification condition can be augmented to ensure that the ranking function strictly decreases with the execution of each program block. Since there are no infinite strictly decreasing sequences of ordinals, this eliminates the possibility of a valid infinite execution on the flowchart.

Hoare [41] transformed Floyd's method into a set of inference rules (see Sect. 20.2.5) for Hoare triples of the form $\{P\}S\{Q\}$ for a well-structured program. The triple expresses the claim that any terminating execution of program S from a state satisfying the assertion P yields a state satisfying the assertion Q. The Floyd–Hoare approach to deductive verification can be illustrated with the simple example shown in Fig. 1 of a program that finds the maximal element in an array of non-negative integers.

The program in Fig. 1 consists of a precondition **true**, a *loop invariant*

$$\mathtt{i} \leq \mathtt{N} \wedge \forall(\mathtt{j} < \mathtt{i}) : \mathtt{a}[\mathtt{j}] \leq \mathtt{max}$$

and a post-condition $\forall(\mathtt{j} < \mathtt{N}) : \mathtt{a}[\mathtt{j}] \leq \mathtt{max}$. The loop invariant, which we abbreviate as $\mathtt{P}(\mathtt{i})$, asserts that the value of the index variable $\mathtt{i}$ is at most $\mathtt{N}$, and all the elements preceding the $\mathtt{i}$th element in the array $\mathtt{a}$ are at most $\mathtt{max}$.

Informally, termination for the program in Fig. 1 can be established by associating a ranking function $\mathtt{N} - \mathtt{i}$ with the `while` loop. If S abbreviates the loop body $\mathtt{if(a[i]} > \mathtt{max)\{max} = \mathtt{a[i]};\}\mathtt{i}{+}{+}\}$, then the triple $\{\mathtt{r} = \mathtt{N} - \mathtt{i}\}S\{\mathtt{r} > \mathtt{N} - \mathtt{i}\}$ is valid since $\mathtt{i}$ is incremented in S. Note that $\mathtt{r}$ is a logic variable in the triple that is implicitly universally quantified. Since this loop satisfies the invariant $\mathtt{N} - \mathtt{i} \geq 0$, the value of the ranking function is always a non-negative integer and the ordering is well founded. Next, we want to establish the post-condition that $\forall(\mathtt{j} < \mathtt{N}) : \mathtt{a}[\mathtt{j}] \leq \mathtt{max}$. This can be derived from a combination of the *loop exit* condition $\neg(\mathtt{i} < \mathtt{N})$ and the loop invariant $\mathtt{P}(\mathtt{i})$. The loop invariant holds trivially

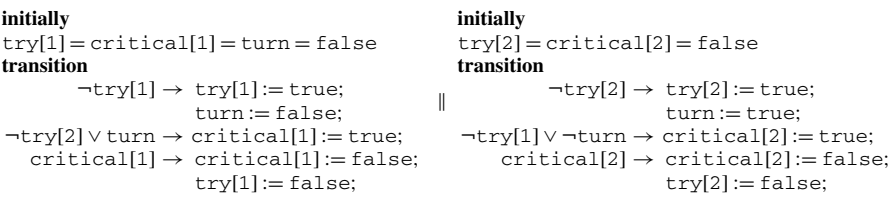


Fig. 2 A two-process mutual exclusion protocol

following the initialization, as expressed by the triple

$$\{\textbf{true}\}\texttt{max} = 0; \texttt{i} = 0\{\texttt{P(i)}\}.$$

The preservation of the invariant by the loop body is expressed by the triple

$$\{\texttt{P(i)} \wedge \texttt{i} < \texttt{N}\}S\{\texttt{P(i)}\}.$$

The latter triple generates the proof obligations below, which are both easily proved.

1. $\texttt{i} < \texttt{N} \wedge \texttt{a[i]} > \texttt{max} \wedge \texttt{P(i)} \Rightarrow \texttt{i} + 1 \leq \texttt{N} \wedge \forall(\texttt{j} < \texttt{i} + 1) : \texttt{a[j]} \leq \texttt{a[i]}$, and
2. $\texttt{i} < \texttt{N} \wedge \texttt{a[i]} \leq \texttt{max} \wedge \texttt{P(i)} \Rightarrow \texttt{P(i} + 1\texttt{)}$.

In the above example, the invariant is a straightforward generalization of the post-condition. To see where the deductive approach is less handy, we examine a simple variant of Peterson's mutual exclusion algorithm [65] represented in the syntax of Unity [22] in Fig. 2. The computation state consists of the control state encoded by two Boolean variables per process: `critical[i]` and `try[i]` with $\texttt{i} = 1, 2$, and a shared Boolean variable `turn`. Each step of the computation applies a transition rule of one of the processes, where each rule is a guarded command. Thus, each computation step is either a step of process 1 according to one of its transition rules (leaving the values of `try[2]` and `critical[2]` unchanged), or a step of process 2 according to one of its transition rules (leaving the values of `try[1]` and `critical[1]` unchanged).

The mutual exclusion property $\neg(\texttt{critical[1]} \wedge \texttt{critical[2]})$ is an invariant in that it holds in every reachable state, but it is not an *inductive* invariant since it is not preserved by the second transition of either process. A stronger invariant asserting

$$(\texttt{critical[1]} \Rightarrow \texttt{turn}) \wedge (\texttt{critical[2]} \Rightarrow \neg\texttt{turn})$$

does turn out to be inductive. Finding such invariant strengthenings is not always simple, and is an active area of research.

Algorithms like those for mutual exclusion need not terminate, but they are required to make progress. For instance, in any computation, once `try[i]` is `true`, then $\texttt{critical[i]} = \texttt{true}$ must eventually hold. Such an eventuality would not follow if, for example, the other process remains in its critical section by never executing the third transition even when the guard condition holds,

i.e., the transition is *enabled*. *Fairness conditions* can be used to ensure that enabled transitions are eventually executed, for example, by requiring the condition `critical[i] = false` to hold infinitely often for each `i` along an execution. Deductive rules based on ranking functions can be given to ensure that `try[i] = true` leads to `critical[i] = true`, but such proofs need to be composed from several eventuality and invariant claims [22, 51].

Model checking [33, 69] was introduced in the early 1980s as an approach for analyzing systems with a lot of control complexity. Here, the flowcharts and programs are less structured and deductive approaches become unwieldy. Examples of such system include hardware, network protocols, and concurrent systems. In the model-checking approach, the transition system is viewed as a model for the property. Such a model can be seen as a directed graph where the states, which assign values to variables, are vertices, and the edges are possible transitions between states given by the program. The verification of properties can be performed by graph exploration to determine whether some class of states or whether a certain kind of cycle is reachable. For the case of the mutual exclusion algorithm in Fig. 2, it is possible to scan the entire reachable portion of the graph in the five Boolean variables to check that there are no violations of the mutual exclusion property. If this check fails, model checking can produce a counterexample in the form of a computation path that leads to a violation of mutual exclusion. Conversely, one could start with the set of "bad" states, i.e., those where $(\texttt{critical[1]} \wedge \texttt{critical[2]})$ holds, and compute the backward reachable states, i.e., those that have computations leading to bad state, to see that no initial state has a computation leading to a bad state. The progress property can also be verified by showing that there are no fair execution paths in the graph where `try[i] = true` does not eventually lead to `critical[i] = true`. Since this is a finite-state system, the only way this eventuality can fail is if the graph contains a state where for some `i`, `try[i] = true` holds and from which it is possible to reach a cycle in which `critical[i] = false` in each state. Instead of a cycle, it is sufficient (and necessary) to find a *strongly connected component*, that is, a subset of states in which there is a path between any two states. Such a strongly connected component is fair if each fairness condition holds for some state in it.

Model-checking problems are decidable for finite-state systems, namely those with state spaces of bounded cardinality, and also for certain extensions to systems with unbounded state spaces. Examples of such extensions are covered elsewhere in this Handbook, and include

1. Timed automata [14] (Bouyer et al., Model Checking Real-Time Systems), cf. [2]
2. Certain limited classes of hybrid automata [31] (Doyen et al., Verification of Hybrid Systems)
3. Parametric systems [1] (Abdulla et al., Model Checking Parameterized Systems)

Abstraction techniques can be used to approximate large or possibly unbounded state spaces by models with small, finite state spaces, and these are covered elsewhere in this Handbook [30] (Dams and Grumberg, Abstraction and Abstraction Refinement), [44] (Jhala et al., Predicate Abstraction for Program Verification).

Compositional techniques for decomposing the verification of composite systems with multiple modules into properties associated with the individual modules are covered elsewhere in this Handbook [36] (Giannakopoulou et al., Compositional Reasoning).

In model checking, an *explicit* representation of the state space is one where a state is represented by a specific assignment of values to variables. Model checking with explicit representations is covered elsewhere in this Handbook [42] (Holzmann, Explicit-State Model Checking). In contrast, symbolic representations of the state space use compact data structures to represent sets of states as formulas. Model checking based on symbolic representations is covered elsewhere in this Handbook [21] (Chaki and Gurfinkel, BDD-Based Symbolic Model Checking). Techniques based on model checking can also be used to synthesize systems for a given property, and this topic is also covered in this Handbook [12] (Bloem, Graph Games and Reactive Synthesis).

In this chapter, we focus on the relationship between deduction and model checking. We first outline the shared background of logic for both approaches. We then examine specific ways in which these approaches can complement each other.

20.2 Logic Background

Logic is a system of notations and rules for making statements, and for proving or refuting these statements. The *syntax* of the logic provides rules for forming well-formed statements. The *semantics* defines the intended meaning of the language primitives by circumscribing their possible interpretations. The *inference rules* of the logic specify how *valid* statements, i.e., those that hold in all possible interpretations, are derived. Different logics correspond to variations in the syntax, semantics, and inference rules. We use propositional logic to illustrate some of the key concepts that are relevant to formal verification.

20.2.1 Propositional Logic

Syntax and Semantics. In propositional logic, statements P and Q are built as *formulas* from propositional atoms such as p and q drawn from a signature Σ. A Σ-formula is constructed from the propositional atoms in Σ using propositional *connectives* such as negation $\neg P$, conjunction $P \wedge Q$, disjunction $P \vee Q$, implication $P \Rightarrow Q$, and equivalence $P \iff Q$. The classical semantics is defined by a truth assignment M that maps propositional atoms in Σ to *truth values*, either $\bot$ for logical falsity or $\top$ for logical truth. The interpretation $M[\![P]\!]$ of a formula P is constructed from the interpretation of its constituents. We let $atoms(P)$ be the set of propositional atoms appearing in P. For a propositional atom p in Σ, the interpretation $M[\![p]\!]$ is its truth assignment $M(p)$. For compound formulas $P \bowtie Q$, where

Fig. 3 One-sided sequent calculus for propositional logic

Ax	$\dfrac{}{\vdash p, \neg p, \Delta}$
$\neg\neg$	$\dfrac{\vdash P, \Delta}{\vdash \neg\neg P, \Delta}$
$\vee$	$\dfrac{\vdash P, Q, \Delta}{\vdash P \vee Q, \Delta}$
$\neg\vee$	$\dfrac{\vdash \neg P, \Delta \quad \vdash \neg Q, \Delta}{\vdash \neg(P \vee Q), \Delta}$
Cut	$\dfrac{\vdash P, \Delta \qquad \vdash \neg P, \Delta}{\vdash \Delta}$

$\bowtie$ is either $\wedge$, $\vee$, $\Rightarrow$, or $\Longleftrightarrow$, the interpretation $M[\![P \bowtie Q]\!]$ is computed from that of $M[\![P]\!]$ and $M[\![Q]\!]$ from the *truth table* interpretation of $\bowtie$. A formula P is *satisfiable* if there is some interpretation M such that $M[\![P]\!] = \top$. We then say that $M \models P$ or M is a *model* of P. If $M \models P$ for all interpretations M, then P is said to be *valid*. The negation of an unsatisfiable formula is valid. A set of formulas Γ is satisfiable if there is a model M such that $M \models P$ for each $P \in \Gamma$, or $M \models \Gamma$, for short.

The model-checking problem for propositional logic is that of checking $M \models P$ for a given M and P. This problem is complete for alternating logarithmic time (ALOGTIME) [20]. The satisfiability problem is that of determining whether there is a model M for a given P. By the celebrated Cook–Levin theorem, this problem is NP-complete [5]. Algorithms for propositional satisfiability are discussed elsewhere in this Handbook [52] (Marques-Silva and Malik, Propositional SAT Solving).

Proof Systems. The inference rules for classical propositional logic can be presented in a number of formats: Hilbert system, natural deduction, or sequent calculus. A proof system for propositional logic based on one-sided *sequents* is shown in Fig. 3. Each sequent has the form $\vdash \Gamma$, where Γ is a (possibly infinite) set of formulas, and represents a *judgement* that under any interpretation, one of the sequent formulas is logically true. The set obtained from adding P to Γ is P, Γ. The axiom rule Ax asserts that any sequent containing a positive and negative atom is provable. Each rule asserts that the conclusion sequent is valid if the premise sequents are. The implication $P \Rightarrow Q$ can be written as $\neg P \vee Q$ and the conjunction $P \wedge Q$ as $\neg(\neg P \vee \neg Q)$. Note that the provability of $P \Rightarrow Q$ can be represented by the sequent $\vdash \neg P, Q$. A formula P is provable if $\vdash P$ can be derived from the axioms given by the Ax rule by applying the rules $\neg\neg$, $\vee$, $\neg\vee$, or Cut. For example, the sequent $\vdash \neg\neg p \vee \neg p$ can be shown to be provable.

Propositional logic is also expressive enough to capture constraints over domains of bounded size. Such domains can be encoded in binary form. Bit vectors of length n can be written as n Boolean variables. An element of a subrange of integers of size n can be encoded by a bit vector of length $\lceil \log_2 n \rceil$. Arrays of at most m elements drawn from a set of cardinality at most n can be represented by $m\lceil \log_2 n \rceil$ bits. Bounded length lists from a bounded set of elements can be represented by arrays. Sets, functions, and relations over bounded domains can also be represented

by Boolean formulas, as can images of functions and relations with respect to sets and compositions of functions and relations. Computations of bounded length over bounded state spaces can also be represented as Boolean constraints. Such an encoding can be used for *bounded model checking*, that is, to determine whether there are computations of a bounded length that violate a specific property.

Propositional logic has a number of useful properties. It is possible to provide proof systems for it that are *sound* (all provable statements are valid), and *complete* (all valid statements are provable). Propositional logic is also *compact*: if a set of formulas is unsatisfiable, there is some finite subset that is already unsatisfiable. For a set of propositional Σ-formulas Γ, we say that a truth assignment M is a model of Γ if it is a model of each formula in Γ. Then Γ is said to *entail* a propositional formula P if every model of Γ is a model of P.

Soundness. A proof system such as the one in Fig. 3 is sound if it proves only valid statements. This means that the sequent $\vdash P$ is provable only when P is valid. More generally, $\vdash \Delta$ is provable only when $\vdash \Delta$ is valid, i.e., for each interpretation M, there is a $P \in \Delta$ where $M \models P$. Soundness can be established by induction on derivations: the *Ax* rule is valid, and the conclusion of each application of an inference rule is valid if its premises are.

Completeness. The sequent calculus in Fig. 3 is complete: every valid sequent is provable. Conversely, if $\vdash P$ is not provable, then we can construct a model M of $\neg P$. A set of formulas Γ is *consistent*, i.e., $Con(\Gamma)$ iff there is no formula P in Γ such that $\vdash \overline{\Gamma}, \neg P$ is provable, where $\overline{\Gamma}$ is the set $\{\neg Q | Q \in \Gamma\}$. By soundness, when $\vdash \overline{\Gamma}, \neg P$ is provable, then $\neg P$ is entailed by Γ since in every interpretation, either Γ is falsified or P is. Note that consistency is a property of the proof system and not a semantic property. If Γ is consistent, then $\Gamma \cup \{P\}$ is consistent iff $\vdash \overline{\Gamma}, \neg P$ is not provable. If Γ is consistent, then at least one of $\Gamma \cup \{P\}$ or $\Gamma \cup \{\neg P\}$ must be consistent. A set of formulas Γ is *complete* if for each formula P, it contains P or $\neg P$. It is not difficult to construct an enumeration $\langle Q_i | i \geq 0 \rangle$ of all the Σ-formulas. With this enumeration, if $\vdash P$ is not provable, we construct a complete set Γ, where $\Gamma = \bigcup_i \Gamma_i$, $\Gamma_0 = \{\neg P\}$, and $\Gamma_{i+1} = \Gamma_i \cup \{Q_i\}$ if $Con(\Gamma_i \cup \{Q_i\})$, and $\Gamma_{i+1} = \Gamma_i \cup \{\neg Q_i\}$, otherwise. For any atom p, define $M_\Gamma(p) = \top$, if $p \in \Gamma$, and $M_\Gamma(p) = \bot$, otherwise. Then, $M_\Gamma \models Q$ for each $Q \in \Gamma$. Hence, $M_\Gamma \models \neg P$. For example, $\vdash p \vee q$ is not provable, and we can construct a model M_Γ starting with $\Gamma_0 = \{\neg(p \vee q)\}$ and checking that Γ contains $\neg p$ and $\neg q$.

Compactness. Propositional logic is compact in the sense that a (possibly infinite) set Γ of Σ-formulas is satisfiable iff it is *finitely satisfiable*, i.e., each finite subset Δ of Γ is satisfiable. Since any subset of a satisfiable set of formulas is satisfiable, we need only show that finite satisfiability implies satisfiability. Any finitely satisfiable set Γ can be extended to a maximal finitely satisfiable set $\widehat{\Gamma}$. This is because, by Zorn's lemma, any partially ordered set Θ in which every linearly ordered subset has an upper bound, contains a maximal element. If we take Θ to be the set of finitely satisfiable extensions of Γ ordered by inclusion, it satisfies the conditions of

Zorn's lemma: the union $\bigcup \mathscr{F}$ of a linearly ordered subset $\mathscr{F}$ of Θ is itself finitely satisfiable since any finite subset of $\bigcup \mathscr{F}$ must already be a subset of an element of $\mathscr{F}$.

For any finitely satisfiable set Δ, either $\Delta \cup \{p\}$ or $\Delta \cup \{\neg p\}$ must be finitely satisfiable. If this were not the case, then for some finite subsets Θ_1 and Θ_2 of Δ, both $\Theta_1 \cup \{p\}$ and $\Theta_2 \cup \{\neg p\}$ are unsatisfiable. In this case, $\Theta_1 \cup \Theta_2$ would already be an unsatisfiable finite subset of Δ, contradicting the finite satisfiability of Δ. Take Δ to be a maximal finitely satisfiable extension $\widehat{\Gamma}$ of a finitely satisfiable set Γ of Σ-formulas. Then, by maximality, for any Σ-atom p, either $p \in \widehat{\Gamma}$ or $\neg p \in \widehat{\Gamma}$. As in the completeness argument, we can define $M_{\widehat{\Gamma}}$ so that $M_{\widehat{\Gamma}}(p) = \top$, if $p \in \widehat{\Gamma}$, and $M_{\widehat{\Gamma}}(p) = \bot$, otherwise. Then $M_{\widehat{\Gamma}}$ must be a model for Γ since if there is a formula $P \in \Gamma$ such that $M_{\widehat{\Gamma}}(P) = \bot$, then the set

$$\left\{p \in \mathit{atoms}(P) \;\middle|\; M_{\widehat{\Gamma}}(p) = \top\right\} \cup \left\{\neg p \in \mathit{atoms}(P) \;\middle|\; M_{\widehat{\Gamma}}(p) = \bot\right\} \cup \{P\}$$

is a finite unsatisfiable subset of $\widehat{\Gamma}$.

By compactness, any unsatisfiable set Γ has a finite subset Δ that is unsatisfiable. Also, if $\Gamma \models P$ then there is a finite subset Δ of Γ such that $\Delta \models P$.

Interpolation. Given a set Γ, a formula P is *incompatible* with Γ if $\Gamma \cup \{P\}$ is unsatisfiable. Given sets Γ_1 and Γ_2 of Σ_1- and Σ_2-formulas, respectively, $\Gamma_1 \cup \Gamma_2$ is unsatisfiable iff there is a Σ_0-formula P that is entailed by Γ_1 and incompatible with Γ_2, where $\Sigma_0 = \Sigma_1 \cap \Sigma_2$. Such a formula P is said to be an *interpolant* [29]. To see why such an interpolant must exist, let Π be the set of Σ_0-formulas P entailed by Γ_1. The argument for the existence of an interpolant has the following steps:

1. If all formulas in Π are compatible with Γ_2, then by compactness there is a Σ_2-interpretation M_2 such that $M_2 \models \Pi \cup \Gamma_2$. Let M be the interpretation M_2 restricted to Σ_0, and define Γ_M to be the set of Σ_0-formulas Q such that $M \models Q$. Note that M is the unique Σ_0-model for Γ_M.
2. If $\Gamma_1 \cup \Gamma_M$ is satisfiable, there must be some Σ_1-interpretation M_1 extending M such that $M_1 \models \Gamma_1$. We then have interpretations M_1 and M_2 such that $M_1 \models \Gamma_1$ and $M_2 \models \Gamma_2$, where M_1 and M_2 restricted to Σ_0 are both equal to M. We can let the interpretation $N(p)$ be $M_1(p)$, if $p \in \Sigma_1$, and $M_2(p)$, otherwise. Then $N \models \Gamma_1 \cup \Gamma_2$ since N, the *amalgamation* of M_1 and M_2, is identical to M_i when restricted to Σ_i, for $i = 1, 2$. This contradicts the assumption that $\Gamma_1 \cup \Gamma_2$ is unsatisfiable.
3. If $\Gamma_1 \cup \Gamma_M$ is unsatisfiable, then by compactness, there must be a minimal finite subset Δ of Γ_M such that $\Gamma_1 \cup \Delta$ is unsatisfiable. Since Δ is finite, we can construct a formula $\bigwedge \Delta$ that is the conjunction of the formulas in Δ. Then $\neg \bigwedge \Delta$ is a Σ_0-formula entailed by Γ_1, and hence $M \models \neg \bigwedge \Delta$. But, by construction, M is a model of all the formulas in Γ_M, and hence all the formulas in Δ. Hence $\Gamma_1 \cup \Gamma_M$ must be satisfiable.
4. Hence, some formula in Π must be incompatible with Γ_2, and is therefore an interpolant.

Fig. 4 Interpolants from cut-free proofs

Ax_1	$\dfrac{}{[\bot] \vdash \Gamma, P, \overline{P}; \Delta}$
Ax_2	$\dfrac{}{[\top] \vdash \Gamma; P, \overline{P}, \Delta}$
Ax_3	$\dfrac{}{[P] \vdash \Gamma, P; \overline{P}, \Delta}$
$\neg\neg$	$\dfrac{[I] \vdash P, \Delta}{[I] \vdash \neg\neg P, \Delta}$
$\vee$	$\dfrac{[I] \vdash A, B, \Delta}{[I] \vdash A \vee B, \Delta}$
$\neg\vee_1$	$\dfrac{[I_1] \vdash \Gamma, \neg A; \Delta \quad [I_2] \vdash \Gamma, \neg B; \Delta}{[[I_1 \vee I_2] \vdash \Gamma, \neg(A \vee B); \Delta}$
$\neg\vee_2$	$\dfrac{[I_1] \vdash \Gamma; \neg A, \Delta \quad [I_2] \vdash \Gamma; \neg B, \Delta}{[I_1 \wedge I_2] \vdash \Gamma; \neg(A \vee B), \Delta}$

The above argument can be extended to first-order logic [35]. However, it merely demonstrates the existence of interpolants without giving a procedure for constructing one. As an example of such a procedure, we show how interpolants can also be constructed from proofs. For example, consider the one-sided sequent calculus for propositional logic containing just negation and disjunction shown in Fig. 3. Here, if we prove a sequent of the form $\vdash \Gamma, \Delta$, then this essentially says that the set of formulas $\overline{\Gamma} \cup \overline{\Delta}$ is unsatisfiable, where $\overline{\Gamma}$ is the set obtained by complementing each of the formulas in Γ. This means there must be some formula P such that $\vdash \Gamma, P$ and $\vdash \neg P, \Delta$. For this, we represent the sequent as $[P] \vdash \Gamma; \Delta$ where P is the interpolant between the two parts Γ and Δ of the sequent separated by a semi-colon. The rules in Fig. 4 show how the interpolant can be constructed for cut-free proofs.

Propositional satisfiability can be made more expressive along several dimensions. Adding quantification over Boolean variables allows first-order logic formulas over bounded domains to be expressed in the logic. The resulting logic fragment, quantified Boolean formulas (QBF), has a PSPACE-complete satisfiability problem. With quantification, there is no distinction between satisfiability and model checking—for a closed formula, i.e., one with no free variables, these are equivalent. Quantified Boolean formulas can be expanded into Boolean formulas with an exponential increase in size. With the added expressiveness of quantification, we can capture image construction, bounded-length games, and inductive relations over bounded domains.

Modal logics [11] offer another dimension of expressibility where the satisfiability of a formula is evaluated relative to a *frame* or Kripke model which is a graph $\langle W, R, L \rangle$ of worlds W that are related by the accessibility relation R and a labeling L mapping worlds to truth assignments for the atomic propositions. The worlds adjacent to or accessible from a given one are the possible alternate truth assignments. The modality $\Box A$ holds in a world if A holds in all worlds that are accessible from it. A statement is valid in a frame if it holds in all of the possible worlds. Many different modal logics can be defined by varying the properties of the accessibility relation. These yield different informal interpretations for the modalities, including necessity, knowledge, belief, and normativity, among others [11, 55].

The model-checking problem for modal logics is typically P-complete, whereas the corresponding satisfiability problem is typically PSPACE-complete.

In the context of formal verification, the accessibility relation corresponding to the progress of time is important. Here the possible worlds correspond to points of time in a computation. One world is accessible from another when it is in the future of the second world. This yields temporal logics [32] where $\Diamond P$ holds in a world when there is some future world where P holds, and $\Box P$ holds if there is no possible future world where $\neg P$ holds. Temporal logic can be further decomposed into linear-time and branching-time logics. In linear time, the possible worlds consist of *paths*, or sequences of states. These paths are related by a suffix relation. A property $\Box P$ holds of a path when P holds of all suffix paths, and a property $\Diamond P$ holds when P holds of some suffix. In contrast, in branching-time temporal logics, to each path representing the future, there are alternate possible paths. The formula $\Box P$ holds of a path if P holds along every suffix of the path, and $\forall P$ holds of a path if P holds of every alternate path. Temporal logics are covered elsewhere in this Handbook [67] (Piterman and Pnueli, Temporal Logic and Fair Discrete Systems). The modal operators can also be indexed by the computations or computation steps. For example, in dynamic logic [39, 68], the modal operators are indexed by programs so that $[\alpha]P$ holds in a state when all computations of the program α from this state terminate in states satisfying P. Hennessy–Milner logic [40] indexes the modal operators $[\alpha]P$ and $\langle\alpha\rangle P$, with the labels α corresponding to the actions in a labeled transition system. We can add fixpoint operators to this calculus to arrive at the modal μ-calculus which is covered elsewhere in this Handbook [15] (Bradfield and Walukiewicz, The mu-calculus and Model Checking). With these operators, we have formulas of the form $\mu X.F[X]$ and $\nu X.F[X]$, where $F[X]$ is a modal formula in which the propositional variable X occurs with positive polarity in $F[X]$, i.e., under an even number of negations.

20.2.2 First-Order Logic

Propositional logic is too limited in its expressivity for many applications. It cannot encode problems over unbounded and infinite domains. Even with problems that can be encoded in propositional logic, one loses the structure and uniformity of the original problem. First-order logic (FOL) [8] refines propositional atoms so that instead of p, we also admit propositions $p(x_1, \ldots, x_n)$, where p is a predicate symbol and $x_1, \ldots, x_n$ are variables. So, instead of an opaque proposition of the form: *Birmingham is the capital of England*, we have a more refined predicate *IsTheCapitalOf*, and we can write formulas containing *IsTheCapitalOf*(x, y). This expressivity is naturally well-suited for mathematical formalization where the domains are typically unbounded and relations abound. FOL can also include function symbols with their associated arity, so that a *term* a is either a variable x or a compound term $f(a_1, \ldots, a_n)$ where f is a function symbol of arity n, with $n \geq 0$. FOL can also contain a primitive equality symbol so that $a = b$ is a formula whenever a and b are

$\neg\exists$	$\dfrac{\vdash \neg P[c/x], \Delta}{\vdash \neg(\exists x.P), \Delta}$
$\exists$	$\dfrac{\vdash P[a/x], \Delta}{\vdash \exists x.P, \Delta}$

Fig. 5 Proof rules for quantification: c must not occur in the conclusion of $\neg\exists$

two terms. If we have a function *Capital* that maps each country to its capital city, then the above proposition can be written as $Birmingham = Capital(England)$.

First-order logic is then defined relative to a signature Σ which is a set containing the allowed function and predicate symbols with their associated arities. FOL formulas are built from equalities $a = b$, atoms $p(a_1, \ldots, a_n)$ for predicate symbol p of arity n, the propositional combinations of negation, disjunction, conjunction, implication, and equivalence, and existential quantification $\exists x.P$ and universal quantification $\forall x.P$. The formula P is the *scope* of the quantifier $\exists x.P$ and $\forall x.P$. An occurrence of a variable x in a formula P is *free* if it does not occur in the scope of a quantifier binding the variable x. The operation of substituting a term a for a free variable x in a formula P is written as $P[a/x]$. Care is required to ensure that such a substitution does not capture free variables in a, so that if y is a free variable in a, then x must not occur within the scope of a quantifier binding y in P. A *sentence* is a closed formula, i.e., one without any free variables. Propositional logic is the fragment of first-order logic where the signature Σ contains only 0-ary predicate symbols.

The semantics for first-order logic over a signature Σ is given by a Σ-structure M with a nonempty set $dom(M)$ (the *domain*) and an interpretation of the function and predicate symbols of Σ as functions and predicates over $dom(M)$. We also need an *assignment* ρ that maps the variables to elements of $dom(M)$ so that the meaning of each term a can be defined as $M[\![a]\!]\rho$. The satisfaction relation $M, \rho \models P$ is defined in the expected manner.

The sequent calculus in Fig. 3 can be extended with a couple of rules for reasoning about existential quantification shown in Fig. 5. The $\neg\exists$ rule allows a sequent formula $\neg P[c/x]$ to be generalized to $\neg\exists P$, provided the constant c does not appear in $\neg(\exists x.P), \Delta$. Since such a constant c can be bound to any element in $dom(M)$ in the premise, the conclusion sequent is justified. The $\exists$ rule allows a formula $\exists x.P$ to be derived from $P[a/x]$ since the term a serves as a witness for the existential quantifier. The rules for universal quantification can be defined from these rules.

First-order logic is a natural formalism for representing mathematical theories. For this purpose, one adds non-logical axioms and axiom schemes to the logic. Mathematical theories that can be represented in this manner include various algebras, Peano arithmetic, and set theory. First-order logic can be given a sound and complete formalization so that all and only the valid statements have proofs. It also has a number of interesting metatheoretic properties like compactness, amalgamation, and interpolation. FOL formulas can be placed in *prenex normal form* where all the quantifiers appear as a prefix at the top of the formula. For example, the sentence $(\forall x.(p(x) \wedge (\exists y.\neg p(y))))$ has the equivalent prenex form $(\forall x.\exists y.p(x) \wedge \neg p(y))$. The existential quantifier can be replaced by a *Skolem function* to get the equisatisfiable formula $(\forall x.p(x) \wedge \neg p(f(x)))$. The resulting formula is unsatisfiable since it

has a *Herbrand expansion* $(p(c) \wedge \neg p(f(c))) \wedge (p(f(c)) \wedge \neg p(f(f(c))))$ which is propositionally unsatisfiable. The Herbrand expansion is a conjunction of instances of the Skolemized formula with terms from the term universe obtained by adding a constant c, if needed, to the Skolem functions. In this case, the Herbrand universe is the set $\{c, f(c), f(f(c)), \ldots\}$. Many proof search methods are based on strategies for constructing unsatisfiable Herbrand expansions.

Certain fragments of first-order logic are decidable [13]. One simple decidable fragment is the first-order theory of pure equality. This theory has an empty signature so that all the atomic formulas are equations or disequations between variables. This fragment is expressive enough for writing formulas that constrain the minimal or maximal number of elements in a model. If a formula in prenex form in this fragment has n distinct variables and is satisfiable, then it is satisfiable in a model with at most n elements. The fragment therefore has the *finite model property*. The monadic predicate calculus with or without equality, where there are no function symbols and only monadic (i.e., arity one) predicate symbols, similarly has the finite model property and is therefore decidable. The Bernays–Schönfinkel fragment consists of prenex formulas of the form $\exists \overline{x}.\forall \overline{y}.A$, where A is a function-free, quantifier-free formula. The satisfiability of sentences in this fragment is decidable. Such a formula, say $\exists x_1, \ldots, x_m.\forall y_1, \ldots, y_n.A$ can be replaced with $\bigvee_{c_1 \in K} \cdots \bigvee_{c_m \in K} \bigwedge_{d_1 \in K} \cdots \bigwedge_{d_n \in K} A[c_1/x_1, \ldots, c_m/x_n, d_1/y_1, \ldots, d_n/y_n]$, where $K = \{a_1, \ldots, a_m\}$. The new formula can be checked for propositional satisfiability.

A *first-order theory* T is the set of sentences that are valid over a given set of interpretations $\mathscr{K}$ so that $T = \{P | \forall M \in \mathscr{K}.M \models P\}$. Such a theory could also be given by a set of sentences closed under consequence that has at least one model. A theory is *stably infinite* if whenever a formula has a model, it has one with a countably infinite domain. The first-order theory of pure equality over infinite models supports *quantifier elimination*: from any formula P, one can construct an equisatisfiable formula $\widehat{P}$ that has no quantifiers. As noted earlier, it is easy to write a sentence that asserts that there are at most k elements, and this sentence has no quantifier-free counterpart. The theory of arithmetic over 0, 1, and + (but without multiplication), known as Presburger arithmetic, also supports quantifier elimination. The validity of a sentence in this theory can therefore be decided by constructing its quantifier-free counterpart and evaluating its validity directly. Other first-order theories that support quantifier elimination include dense linear orders, algebraically closed fields, and real closed fields.

Within theories, one can look at whether it is decidable to check for the validity of formulas of a specific form. For a formula P, let $\forall P$ represent the universal closure $\forall \overline{x}.P$ of P, where $\overline{x}$ is a sequence of the free variables in P. The *word problem* for a theory is that of deciding the validity of a formula $\forall P$, where P is atomic. The *uniform word problem* is that of deciding the validity of a formula $\forall(P_1 \wedge \cdots \wedge P_n \Rightarrow P)$, where P and the formulas P_i, for $1 \leq i \leq n$, are atomic. The *clausal validity problem* is that of deciding the validity of formulas of the form $P_1 \vee \cdots \vee P_n$, where each P_i is either an atomic formula or the negation of an atomic formula. A theory is *convex* when a clause $\neg p_1 \vee \cdots \vee \neg p_m \vee q_1 \vee \cdots \vee q_n$ is valid iff the uniform word problem $\neg p_1 \vee \cdots \vee \neg p_m \vee q_i$ is valid, for some i,

$1 \leq i \leq n$. For example, the theory of linear arithmetic over the reals is convex, whereas the theory of integer linear arithmetic is non-convex since $x > 1 \wedge x < 5 \Rightarrow (x = 2 \vee x = 3 \vee x = 4)$ without the antecedent implying any one of the consequent disjuncts. Since any quantifier-free formula P can be expressed as a conjunction of clauses $K_1 \wedge \cdots \wedge K_n$, the validity of $\forall P$ can be reduced to the validity of $\forall K_i$ for each i, $1 \leq i \leq n$. However, a more efficient approach to the quantifier-free validity problem is to use clausal validity within a Boolean satisfiability (SAT) procedure to detect that a partial assignment computed by the SAT solver is unsatisfiable in a theory. Theory solvers (for clausal validity) can also be used to strengthen Boolean constraint propagation to exploit theory propagation so that, for example, when $f(x) \neq f(y)$ is in the partial assignment, the literal $x = y$ is immediately implied as being false. The combination of Boolean satisfiability checking with theory solving, namely Satisfiability Modulo Theories (SMT), is covered elsewhere in this Handbook [7] [Barrett and Tinelli, Satisfiability Modulo Theories).

A formula is satisfiable in a *combination* of theories T_1 over a signature Σ_1 and T_2 over a signature Σ_2 if it has a $(\Sigma_1 \cup \Sigma_2)$-model M such that M restricted to Σ_i is a T_i-model, for $i = 1, 2$. The Nelson–Oppen method [58] can be used to demonstrate clausal validity in a combination of theories with disjoint signatures. A clause K is first purified to an equisatisfiable form $K_1 \vee K_2$, where each K_i is a Σ_i-clause, for $i = 1, 2$. If the negation $\overline{K_1} \wedge \overline{K_2}$ is unsatisfiable, then there is an interpolant formula Q in $\Sigma_1 \cap \Sigma_2$ such that $\overline{K_1}$ entails Q and $Q \wedge \overline{K_2}$ is unsatisfiable. Since Q is in the empty theory and over the shared variables in $\overline{K_1}$ and $\overline{K_2}$, it is in the theory of pure equality. If the theories T_1 and T_2 are stably infinite, then by quantifier elimination, there must be a quantifier-free interpolant formula Q in the free variables shared by K_1 and K_2. Such a Q can be constructed by testing each possible arrangement which is a partition of the free variables into equivalence classes. Each such arrangement can be represented as a conjunction of equalities and disequalities. For example, the conjunction $x = y \wedge y = z \wedge x \neq u$ represents two equivalence classes: $\{x, y, z\}$ and $\{u\}$. Then $\overline{K_1} \wedge \overline{K_2}$ is unsatisfiable if for each arrangement C, either $\overline{K_1} \wedge C$ or $\overline{K_2} \wedge C$ is unsatisfiable. Each of the latter checks for unsatisfiability uses the satisfiability procedure for the individual theories. The interpolant Q is $\bigvee\{C | \overline{K_1} \wedge C$ is satisfiable$\}$ since every model of $\overline{K_1}$ satisfies some disjunct in Q.

Model checking first-order logic formulas relative to finite models is a PSPACE-complete problem. Note that it subsumes QBF satisfiability. The complexity of checking a model for a fixed formula with respect to the size of the model, i.e., the *model complexity*, corresponds to AC^0, the class of bounded-depth circuits of polynomial size. This problem is related to database query evaluation and constraint solving, and plays a central role in descriptive complexity theory [43]. Modal logics can be translated to first-order logic by introducing an explicit accessibility relation. However, first-order logic is not expressive enough to capture concepts such as finiteness, the transitive closure of a relation, or fixpoints of monotone predicate transformers. Adding least and greatest fixpoints (covered elsewhere in this Handbook [15] (Bradfield and Walukiewicz, The mu-calculus and Model Checking)) to first-order logic enhances the expressiveness, but is still not enough to capture finiteness [64].

20.2.3 Higher-Order Logic

Some of the limitations in the expressiveness of first-order logic can be overcome in higher-order logic (HOL) [3, 48], which extends quantification to functions and predicates. This kind of quantification is already present in first-order logic since the predicate and function symbols in a formula are assumed to be implicitly universally quantified. To avoid inconsistencies arising from paradoxes due to the self-application of predicates to predicates, higher-order logic is typed according to a strict hierarchy of types. Otherwise, we could have a predicate R that is defined so that $R(X) = \neg X(X)$, and we have an inconsistency when X is instantiated with R itself. Modern higher-order logics are based on Church's simple theory of types [23]. The types are built from the base types of individuals ι and propositions o using the function type constructor $\tau_1 \to \tau_2$ which is the type of maps from elements of type τ_1 to type τ_2. Higher-order logic formulas can be constructed from equality operators of type $\tau \to \tau \to o$ (parsed as $\tau \to (\tau \to o)$) for each type τ, using *application* $s\ t$ and *lambda abstraction* $\lambda(x : \tau) : s$. A quantified formula $\forall(x : \tau).P$ can be defined as $(\lambda(x : \tau).P) = (\lambda(x : \tau).\top)$, where $\top$ is itself defined using prefix notation as $= (=)(=)$, where the second and third occurrences of '$=$' have type $\iota \to \iota \to o$, and the first occurrence has type $[\iota \to \iota \to o] \to [\iota \to \iota \to o] \to o$. A type τ has order 1 if it is either ι or o, and it has order $i + 1$ if it is of the form $\tau_1 \to \tau_2$, where τ_1 has order at most i and τ_2 has order at most $i + 1$. An ith-order logic admits quantification over variables with types of ith order. Thus second-order logic allows quantification over variables of type $\iota \to o$ and $\iota \to \iota \to \iota$ in addition to quantification over first-order types.

Sets with elements of type τ in higher-order logic are just predicates on τ which have the type $\tau \to o$. The subset ordering on sets can also be easily defined. With this, the set of natural numbers can already be defined in second-order logic as the least set containing 0 that is closed under the successor operation. Finiteness is easily expressed since a set is finite iff every injective map on it is surjective, or equivalently, if there is no bijection between the set and any proper subset. The definition of natural numbers is an instance of a fixpoint definition. The least and greatest fixpoint operators μ and ν can be defined in higher-order logic using the Knaster–Tarski theorem. Compared to first-order logic, modal logics can be semantically embedded in a higher-order logic by directly defining the modal operators. Model-checking queries can be expressed with this embedding and model checkers can be used as decision procedures for such queries [70].

Higher-order logic lacks many of the metatheoretic properties of first-order logic. Since it can be used to define natural numbers, higher-order logic is not complete with respect to the standard semantics of function types $\tau_1 \to \tau_2$ as the set of all maps from the set interpreting τ_1 to the one interpreting τ_2. It is complete with respect to the more liberal Henkin interpretations. Properties like compactness and interpolation fail even in second-order logic. However, higher-order logic offers greater convenience for formalizing and manipulating concepts in mathematics and computing, and for embedding various logics and theories. Furthermore, the useful properties of first-order logic can always be invoked for the first-order fragment of higher-order logic.

```
bubble[N : nat] : THEORY
  BEGIN

  ARR: TYPE = [below(N) -> nat]

  A, B : VAR ARR

  m: VAR below(N)

  max(A, m): bool = (FORALL (j : upto(m)): A(j) <= A(m))

  bubble(A, m, (i : upto(m) | max(A, i))): RECURSIVE {B | max(B, m)} =
   (IF i = m
     THEN A
     ELSIF A(i) > A(i+1)
      THEN bubble(A WITH [(i) := A(i+1), (i+1) := A(i)], m, i+1)
      ELSE bubble(A, m, i+1)
    ENDIF)
    MEASURE m - i

  END bubble
```

Fig. 6 A PVS Theory

20.2.4 PVS: Computation and Deduction

Logics are typically designed as idealized objects of mathematical study rather than for actual use in formal modeling and reasoning. Further features are needed to support facile expression and efficient proof construction. To illustrate some of the pragmatic aspects of using logic, we examine a few of the language and deductive features of the Prototype Verification System (PVS), a specification and verification framework based on higher-order logic [63]. The specification language extends higher-order logic with polymorphic equality, conditionals, and updates, as well as with predicate subtypes, dependent types, parametric theories, and theory interpretations. These features, several of which were originally introduced in the EHDM specification language [74], bring the language closer to a mathematical vernacular than the textbook presentations of first-order and higher-order logic.

The language and deductive features in PVS can be illustrated by the example in Fig. 6. The theory `bubble` takes a parameter `N` that is declared to be a natural number which is itself a subtype of the integers defined as `{i : int | i >= 0}`. The subtype `below(N)` is a possibly empty parametric predicate subtype consisting of the natural numbers in the subrange from `0` to `N-1`. The array type `ARR` is declared to map indices in `below(N)` to the natural numbers. An index variable `m` ranges over `below(N)`. The predicate `max` is defined to check that `m` is the index of the maximal element of `A` for indices in the subrange from `0` to `m`. The `bubble` operation takes as input an array `A`, a bound `m` in `below(N)`, and an index `i` in the range between `0` and `m` such that `A(i)` is the maximal element among the elements from `A(0)` to `A(i)` in the array. The operation is declared to return an array whose maximal element is in position `m`. The type of `i` *depends* on the parameter `m`, as does the return type `{B | max(B, m)}`. `bubble` is defined recursively to

successively increment `i` and to swap the elements `A(i)` and `A(i+1)` if `A(i) > A(i+1)` at each position `i`. Note that the identifier `bubble` is used to name the theory and the operation, and PVS is quite liberal about such overloading. Typechecking the theory `bubble` generates nine proof obligations or *type correctness conditions* (TCCs) covering subtyping and termination. Automated proof strategies in PVS allow these proof obligations to be easily discharged. In the first couple of attempts, there typically are unprovable TCCs corresponding to specification errors.

The features of PVS that are used in `bubble` illustrate the pragmatic aspects of formal logic. Higher-order logic is flat and offers no features for structuring specifications into theories. In PVS, *parametric theories* are used to package axioms, definitions, and theorems in a reusable form. Examples of such packages include algebras such as groups, parametric datatypes such as lists over an element type, and polymorphic algorithms. *Theory interpretations* allow types and symbols in abstract theories to be instantiated by concrete interpretations that satisfy the axioms. Such interpretations can be used to demonstrate consistency or as a way of reusing an abstract development, such as a theory of groups, on a concrete group. Predicate subtypes are quite customary in mathematical vernacular. For example, the even numbers are a subtype of the integers, which are in turn a subtype of the rational numbers, and the latter are a subtype of the real numbers. Subtyping not only includes first-order subtypes such as nonzero numbers (needed to ascribe a type to the denominator of the division operation), subranges, prime numbers, and Mersenne primes, but also extends to higher-order concepts like monotone maps, injections, continuous functions, and group homomorphisms. Typechecking with subtypes detects many significant errors and even gaps such as, for example, claiming that $\frac{n!}{(n-k)!k!}$ is an integer without proper justification. Significant fragments of the PVS language are executable, and typechecking ensures that the execution is both efficient and free of runtime errors other than those arising from resource bounds.

Proofs in PVS are constructed interactively by invoking proof commands on a goal to generate subgoals. These commands can either be primitive steps, including those that invoke external simplifiers such as SAT/SMT solvers, a Boolean simplifier using Binary Decision Diagrams (BDDs), a μ-calculus model checker, and a predicate abstractor, or they are compound proof strategies that can be defined by the end-user. Other proof assistants that offer practical support for proof construction include ACL2 [46], Coq [9], HOL [38], Isabelle [62], and Nuprl [25]. Liquid type systems [73] explore type inference with predicate subtypes using predicate abstraction (see Sect. 20.3.3).

20.2.5 Hoare Logic

As mentioned in Sect. 20.1, a Hoare triple has the form $\{P\}S\{Q\}$, where P and Q are assertions containing logical variables and program variables, and S is a program statement. The logic variables are drawn from a set X and the program variables from a finite set Y, Σ is a first-order signature, C ranges over $\Sigma[Y]$-formulas,

Fig. 7 A Hoare Calculus

Skip	$\{P\}skip\{P\}$
Assignment	$\{P[\overline{e}/\overline{y}]\}\overline{y} := \overline{e}\{P\}$
Conditional	$\dfrac{\{C \wedge P\}S_1\{Q\} \quad \{\neg C \wedge P\}S_2\{Q\}}{\{P\}C\ ?\ S_1\ :\ S_2\{Q\}}$
Loop	$\dfrac{\{P \wedge C\}S\{P\}}{\{P\}while\ C\ do\ S\{P \wedge \neg C\}}$
Composition	$\dfrac{\{P\}S_1\{R\} \quad \{R\}S_2\{Q\}}{\{P\}S_1; S_2\{Q\}}$
Consequence	$\dfrac{P \Rightarrow P' \quad \{P'\}S\{Q'\} \quad Q' \Rightarrow Q}{\{P\}S\{Q\}}$

$\overline{e}$ ranges over sequences of n $\Sigma[Y]$-terms, and S ranges over program statements, where a program statement is one of

1. A *skip* statement *skip*.
2. A *simultaneous assignment* $\overline{y} := \overline{e}$ where $\overline{y}$ is a sequence of n distinct program variables.
3. A *conditional* statement $C\ ?\ S_1\ :\ S_2$.
4. A *loop while C do S*.
5. A sequential composition $S_1; S_2$.

If P, Q, R range over program assertions which are $\Sigma[X \cup Y]$-formulas, then the inference rules of the Hoare calculus are as shown in Fig. 7.

The semantics can be given relative to a Σ-structure M which provides the interpretation of the symbols in Σ. A state σ maps each program variable y in Y to a value in $dom(M)$. A $\Sigma[Y]$-expression e has the value $M[\![e]\!]\sigma$ where $M[\![y]\!]\sigma = \sigma(y)$ and $M[\![f(e_1, \ldots, e_n)]\!]\sigma = M(f)(M[\![e_1]\!]\sigma, \ldots, M[\![e_n]\!]\sigma)$. The meaning $M[\![S]\!]$ of a statement S is given by a set of sequences (of length at least two) of states, and it is defined inductively as shown below. Here, ψ range over such a sequence of states of the form $\psi[0], \ldots, \psi[n-1]$. The singleton sequence consisting of state σ is written as σ itself. The sequence $\psi_1 \circ \psi_2$ is the concatenation of ψ_1 and ψ_2.

1. $\sigma \circ \sigma \in M[\![skip]\!]$, for any state σ.
2. $\sigma \circ \sigma[M[\![\overline{e}]\!]\sigma/\overline{y}] \in M[\![\overline{y} := \overline{e}]\!]$, for any state σ.
3. $\psi_1 \circ \sigma \circ \psi_2 \in M[\![S_1; S_2]\!]$ for $\psi_1 \circ \sigma \in M[\![S_1]\!]$ and $\sigma \circ \psi_2 \in M[\![S_2]\!]$.
4. $\psi \in M[\![C\ ?\ S_1\ :\ S_2]\!]$ if either $M[\![C]\!]\psi[0] = \top$ and $\psi \in M[\![S_1]\!]$, or $M[\![C]\!]\psi[0] = \bot$ and $\psi \in M[\![S_2]\!]$.
5. $\sigma \circ \sigma \in M[\![while\ C\ do\ S]\!]$ if $M[\![C]\!]\sigma = \bot$.
6. $\psi_1 \circ \sigma \circ \psi_2 \in M[\![while\ C\ do\ S]\!]$ if $M[\![C]\!](\psi_1[0]) = \top$, $\psi_1 \circ \sigma \in M[\![S]\!]$, and $\sigma \circ \psi_2 \in M[\![while\ C\ do\ S]\!]$.

Note that the semantic definition is deterministic so that for $\psi_1, \psi_2 \in M[\![S]\!]$, if $\psi_1[0] = \psi_2[0]$, then $\psi_1 = \psi_2$. Also, if the execution of S diverges on a program state σ, then there is no $\psi \in M[\![S]\!]$ with $\psi[0] = \sigma$.

Soundness and Completeness. If σ maps variables in Y to $dom(M)$ and ρ maps variables in X to $dom(M)$ then a $\Sigma[X \cup Y]$-formula P is interpreted as $M[\![P]\!]^{\rho}_{\sigma}$,

Fig. 8 A Hoare triple for a program that multiplies and then divides

$\{k \geq 0 \wedge m > 0 \wedge x = k \wedge y = m\}$
$j := 0; i := 0;$
while $i < k$ *do* $\{j := j + m; i := i + 1\};$
$l := 0;$
while $j \geq m$ *do* $\{j := j - m; l := l + 1\}$
$\{x = l\}$

where $M[\![y]\!]^{\rho}_{\sigma} = \sigma(y)$ for $y \in Y$, and $M[\![x]\!]^{\rho}_{\sigma} = \rho(x)$ for $x \in X$. A Hoare triple $\{P\}S\{Q\}$ is *valid* in a Σ-structure M if for every sequence $\sigma \circ \psi \circ \sigma' \in M[\![S]\!]$ and any assignment ρ of values in $dom(M)$ to logical variables in X, either $M[\![Q]\!]^{\rho}_{\sigma'} = \top$ or $M[\![P]\!]^{\rho}_{\sigma} = \bot$. The Hoare calculus is sound relative to any Σ-structure M: every derivable triple is valid in M.

The next step is to show that any valid Hoare triple is derivable. For this, we need the language Σ and its models to be expressive enough to capture the assertions needed to verify triples. The proof of a valid triple $\{P\}S\{Q\}$ can be decomposed into the valid triple $\{wlp(S)(Q)\}S\{Q\}$ and the valid assertion $P \Rightarrow wlp(S)(Q)$, where $wlp(S)(Q)$ (the *weakest liberal precondition*) is the weakest assertion P such that for any $\psi \in M[\![S]\!]$ with $|\psi| = n + 1$ and ρ, either $M[\![Q]\!]^{\rho}_{\psi[n]} = \top$ or $M[\![P]\!]^{\rho}_{\psi[0]} = \bot$.[1] If for each S and Q, there is an R such that $R = wlp(S)(Q)$, then the triple $\{R\}S\{Q\}$ can be verified by the Hoare calculus. This follows by induction on the structure of S. For example, that if $R = wlp(\textit{while } C \textit{ do } S)(Q)$, then the triple $\{R \wedge C\}S\{R\}$ is provable, because $R \wedge C \Rightarrow wlp(S)(R)$, and $R \wedge \neg C \Rightarrow Q$. Either or both of these valid implications, $R \wedge C \Rightarrow wlp(S)(R)$ and $R \wedge \neg C \Rightarrow Q$, might be unprovable because the underlying assertion logic is incomplete. The Hoare logic is therefore *relatively complete* since it reduces the provability of a valid triple to a set of valid proof obligations in the assertion logic [4, 26].

If we pick the assertion logic to be Presburger arithmetic (see Sect. 20.2.2), then one can write programs in this language for which the needed assertions cannot be expressed. Rules like *Composition* and *Consequence* require new assertions to be invented, and these might not be in the same language. For example, Fig. 8 shows a Hoare triple of the form $\{P\}S\{Q\}$, where the program S consists of two consecutive loops where the first loop multiplies a non-negative integer k and a positive integer m by successive addition to compute j, and the second loop divides j by m through successive subtraction to obtain i. The precondition uses two logic variables x and y to bind the initial values of k and m, respectively. The post-condition asserts that the final value of l is the same as the initial value of k which is bound to the logic variable x. The correctness argument for the triple requires intermediate assertions involving multiplication and therefore cannot be conducted in Presburger arithmetic.

If we use the first-order theory of arithmetic over the signature consisting of the constants 0 and 1 and the operations $+$ and $\times$, the first source of incompleteness, namely the inexpressiveness of the assertion language, vanishes. This is not

[1] Note that the *weakest precondition* $wp(S)(Q)$ is a stronger predicate P with the added condition that for any σ such that $M[\![P]\!]_{\sigma} = \top$, there is a $\psi \in M[\![S]\!]$ such that $\psi[0] = \sigma$, i.e., the computation of S executed from the state σ terminates.

merely because we can express multiplication but because we can capture computation. The inductive definition of the set of computation sequences needed for defining $wlp(S)(Q)$ above can be codified in a number of theories, e.g., first-order arithmetic, through Gödel numbering. If we assume that there are n program variables $y_0, \ldots, y_{n-1}$ from Y, then each state can be represented as a sequence of numerals $\underline{k_0}, \ldots, \underline{k_{n-1}}$ representing the state σ when $\sigma(y_i) = k_i$ for $0 \leq i < n$, where $\underline{k_i}$ is the numeral in first-order arithmetic representing the number k_i. The sequence $\underline{k_0}, \ldots, \underline{k_{n-1}}$ representing σ can itself be encoded as a number, as can the sequence of states ψ. Let $\underline{\psi}$ represent the numeral corresponding to the arithmetic encoding of the sequence ψ, so that the operation $first(\underline{\psi})$ and $last(\underline{\psi})$ represent the numerals for the encoding of the first and last states, respectively. For any statement S, we can define an arithmetic predicate p_S such that $M \models p_S(\underline{\psi})$ iff $\psi \in M[\![S]\!]$. With such a predicate, we can formalize the weakest liberal precondition as $wlp(S)(Q) = \forall z.\overline{y} = first(z) \wedge p_S(z) \Rightarrow Q[last(z)/\overline{y}]$, where $\overline{y} = \underline{\sigma}$ captures the assertion $\bigwedge_{i=0}^{n-1} y_i = \underline{k_i}$ when $\underline{\sigma}$ encodes the sequence $\underline{k_0}, \ldots, \underline{k_{n-1}}$, and $Q[last(z)/\overline{y}]$ is the result of substituting $\underline{k_i}$ for y_i in Q when $last(z)$ encodes the sequence $\underline{k_0}, \ldots, \underline{k_{n-1}}$. This definition of *wlp* captures the informal idea that assertion P is the weakest liberal precondition of Q with respect to S if all terminating computations of S from a state satisfying P terminate in a state satisfying Q. A theory such as first-order arithmetic is said to be *expressively complete* since it can express $wlp(S)(Q)$ for any first-order arithmetic assertion Q.

The Hoare calculus above features a simple programming language defined over a state consisting of a finite set of variables. Many extensions have been developed to account for a range of language features. Separation logic is especially interesting since it supports local assertions about heap-allocated structures like arrays, linked lists, and trees [72].

20.3 Deduction and Model Checking

We have outlined the relationship between deduction and model checking going from propositional to higher-order logic. We now examine the interaction between the two approaches on specific verification problems and techniques.

20.3.1 Abstract Interpretation

Abstract interpretation [28, 61] is used to compute program properties such as the possible signs or intervals of values assigned to a variable and shapes of data structures, by using an abstract lattice for approximating program behavior. A lattice is a partially ordered set that is closed under the meet operation $x \sqcap y$ and join operations $x \sqcup y$ which are respectively the greatest lower bound and least upper bound of x and y. A complete lattice is a partially ordered set that contains greatest lower

bounds and least upper bounds of arbitrary subsets. From the Knaster–Tarski theorem, we know that any monotone operator on a complete lattice has a complete lattice of fixpoints, including a least and greatest fixpoint.

Any program semantics can be seen as a concrete lattice, a Boolean lattice where the partial order is the subset ordering. For example, the *trace* semantics for *while*-programs presented in Sect. 20.2.5 can be viewed as a concrete lattice. The trace semantics could itself be abstracted by the set of states that precede each statement in the program. A further abstraction would be to simply consider the set of values that a variable can take. For example, in the program in Fig. 8, the variable k takes only a single value indicated by the logic variable x, the variable i ranges between 0 and x, and the variable j takes on values that are multiples of y between 0 and $x \times y$. The program actions can be mapped to a transfer function F on the lattice. The range of values of a variable, say i, can be computed as the least set I such that $F_i(I) \subseteq I$, where F_i is the transfer function for the variable i. As an approximation, we could also just compute a *pre-fixpoint* by finding some I, not necessarily the least one, such that $F_i(I) \subseteq I$. For example, the initial range of values for the variable k is $\{x\}$, and this remains the range of values for k after each execution. For i, the initial set of values is the empty set, and there are statements that add 0 to it, that add $z+1$ if z is in the range and $z < x$ (given that the range of k is $\{x\}$) so that we can say that the value of i is always in the range $[0, x]$. The latter range could be computed by using the lattice of intervals as an abstract domain. We can also use the sign lattice $\{\ominus, \mathbf{0}, \oplus, \top\}$ discussed below to interpret the program in order to demonstrate that the possible values of j are non-negative.

The computation of an approximation to a fixpoint on a concrete lattice can be mapped to the computation of fixpoints on an abstract lattice through a Galois connection. Let $\langle C, \leq, \bigcap \rangle$ be a concrete lattice and $\langle A, \sqsubseteq, \bigsqcap \rangle$ be an abstract lattice. A Galois connection is a pair (α, γ) of maps: α from C to A, and γ from A to C, such that for any $a \in A$ and $c \in C$, $\alpha(c) \sqsubseteq a \Leftrightarrow c \leq \gamma(a)$. Intuitively, $\gamma(a)$ is the (largest) concretization of a, and $\alpha(c)$ is the (strongest) abstraction for c, on the respective partial orders. Note that $c \leq \gamma(\alpha(c))$ and $\alpha(\gamma(a)) \sqsubseteq a$ for any $c \in C$ and $a \in A$. The maps α and γ are order-preserving. If F is a monotone operator on C, then μF is the least fixpoint of F in C and can be defined as $\bigcap\{X | F(X) \leq X\}$. It is possible to define an abstract operator $\widehat{F} = \alpha \circ F \circ \gamma$ from a concrete operator F. It is easy to see that $\mu F \leq \gamma(\mu \widehat{F})$. Furthermore, if $\widehat{F}(a) \sqsubseteq G(a)$ for all $a \in A$, then $\mu F \leq \gamma(\mu G)$. Also, if Y is a post-fixpoint of $\widehat{F}$, then $\gamma(Y)$ is a post-fixpoint of F and an inductively valid property of the concrete computation.

Deductive techniques for optimization can be used to compute the abstraction α as a step in the construction of the abstract fixpoint [49, 77] or to directly precompute the abstract transfer function [27, 71]. This kind of precomputation can be illustrated using a *sign abstraction* for the integer domain given by an abstract domain $D = \{\mathbf{0}, \oplus, \ominus, \top\}$, where $\gamma(\mathbf{0})$ is the set $\{0\}$, $\gamma(\oplus)$ is the set of non-negative integers $[0, \infty)$, $\gamma(\ominus)$ is the set of non-positive integers $(-\infty, 0]$, and $\gamma(\top)$ is the set of integers $(-\infty, \infty)$. The operations $+$ and $-$ on integers can be lifted to the corresponding operations $\widehat{+}$ and $\widehat{-}$ on D as shown in Fig. 9. The table of entries in Fig. 9 can be precomputed using theorem proving. Each entry is computed by showing

Fig. 9 Abstract versions of $+$ and $-$

$\widehat{+}$	0	$\oplus$	$\ominus$	$\top$
0	0	$\oplus$	$\ominus$	$\top$
$\oplus$	$\oplus$	$\oplus$	$\top$	$\top$
$\ominus$	$\ominus$	$\top$	$\ominus$	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

$\widehat{-}$	0	$\oplus$	$\ominus$	$\top$
0	0	$\ominus$	$\oplus$	$\top$
$\oplus$	$\oplus$	$\top$	$\oplus$	$\top$
$\ominus$	$\ominus$	$\ominus$	$\top$	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

that on the concrete lattice, we have a transformer on the sets c_x and c_y of the possible values of variables x and y, where we define $c_x + c_y$ as $\{u + v | u \in c_x, v \in c_y\}$. Then, we can compute an abstract transfer $\widehat{+}$ by defining $\widehat{x}\widehat{+}\widehat{y}$ to be the least element a in the sign lattice such that $\gamma(\widehat{x}) + \gamma(\widehat{y}) \subseteq \gamma(a)$. The latter condition with, for example, $\ominus$ for a, $\oplus$ for $\widehat{x}$, and $\ominus$ for $\widehat{y}$, corresponds to the proof obligation $\forall x, y.x \geq 0 \wedge y \leq 0 \Rightarrow x + y \leq 0$. This proof obligation is clearly not valid, whereas when a is $\top$, the corresponding proof obligation is valid. Similarly, if the concrete operation is a condition, e.g., $x > 3$, then the transfer function can be computed as $\widehat{x} \sqcap \oplus$. The transfer function computation can be done in order of increasing precision to compute $\oplus\widehat{+}\oplus$ before $\mathbf{0}\widehat{+}\oplus$ so that the latter value already has a useful upper bound. We can also make use of strictness to compute entire rows or columns in one step so that $\widehat{x}\widehat{+}\top = \top\widehat{+}\widehat{x} = \top$ for any $\widehat{x}$.

20.3.2 Symbolic Model Checking

We have already seen that modal and temporal formulas can be model checked with respect to a Kripke model. Such Kripke models represent transition systems that are described in symbolic form as a triple $\langle W, I, N \rangle$, where W is the type of states, I is the initialization predicate on states, and N is a binary transition relation on states. It is often infeasible to compute the concrete Kripke model from the symbolic transition system. For this reason, model checking is carried out with a symbolic representation of the model. The use of a symbolic representation somewhat blurs the distinction between deduction and model checking. Reachability is a canonical model-checking query and it has the μ-calculus form $\mu X.I \sqcup N[X]$. In order to compute the symbolic representation of the set of reachable states, it is enough to have methods for computing

1. The least upper bound $X \sqcup Y$ of two formulas
2. The strongest post-condition given by the image $N[X]$ of a formula X with respect to the relation N, and
3. The equivalence operator $X \equiv Y$ to check when the fixpoint has been reached.

When the symbolic representation is propositional logic, these operations can be efficiently implemented with BDDs, which are covered elsewhere in this Handbook [19] (Bryant, Binary Decision Diagrams), [21] (Chaki and Gurfinkel, BDD-Based Symbolic Model Checking). SAT solvers, covered elsewhere in this Handbook [52] (Marques-Silva and Malik, Propositional SAT Solving), can also be used to compute these operations – for example, the image computation can be carried out

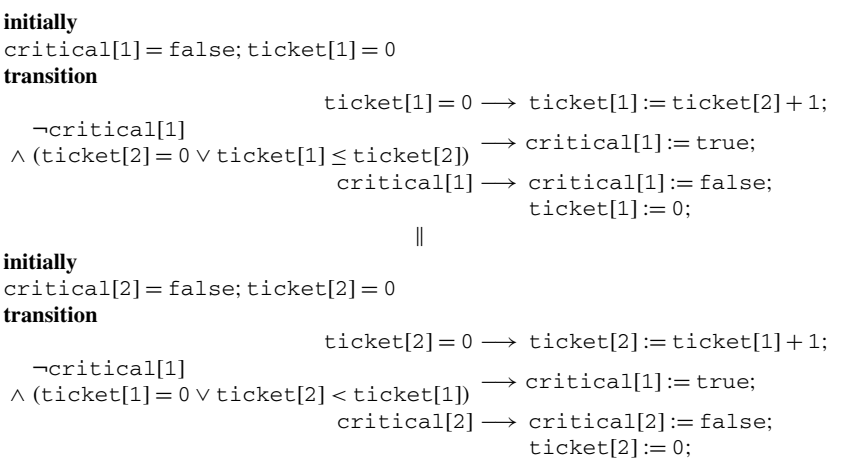


Fig. 10 A two-process Bakery mutual exclusion protocol

using a variant of AllSAT which generates a representation of all possible feasible solutions for a subset of the variables. When the initialization predicate and transition relation are expressible with Boolean combinations of difference constraints, the image computation can be computed using quantifier elimination. Other symbolic representations for model checking are covered in this Handbook [50] (Majumdar and Raskin, Symbolic Model Checking in Non-Boolean Domains).

20.3.3 Predicate Abstraction

In abstract interpretation, fixpoints are computed dynamically on an abstract lattice by precomputed transfer functions on this lattice. Fixpoints such as reachability on infinite-state systems as well as large finite-state systems can be computed by approximating these systems with a smaller system on which we can apply explicit-state or symbolic model checking. In predicate abstraction [75, 76], covered elsewhere in this Handbook [44] (Jhala et al., Predicate Abstraction for Program Verification), we have a concrete transition system $\langle W_C, I_C, N_C \rangle$ and this is approximated by an abstract transition system $\langle W_A, I_A, N_A \rangle$, where bits in the abstract system state W_A represent sets of concrete states, or equivalently, predicates over the concrete states. For a simple example, we use an instance of Lamport's Bakery algorithm shown in Fig. 10. This is actually an infinite-state system since the ticket value is potentially unbounded if each process exiting the critical section races around to grab a ticket while the other process is either in its critical section or waiting to enter it.

Despite being an infinite-state system, the mutual exclusion property can be verified by observing specific predicates that already occur in the transition system, i.e., `critical[1]`, `critical[2]`, `ticket[1]` $= 0$, `ticket[2]` $= 0$,

$\texttt{ticket}[1] \leq \texttt{ticket}[2]$. If we label these five predicates by Boolean variables $\texttt{c}_1, \texttt{c}_2, \texttt{z}_1, \texttt{z}_2, \texttt{p}$, we can build an abstract transition system in these Boolean variables that tracks the behavior of these predicates in the concrete transition system. With this abstraction, the initialization $I_C(1)$ for process 1 becomes $I_A(1) = \texttt{c}_1 = \texttt{false} \wedge \texttt{z}_1 = \texttt{true} \wedge \texttt{p} = \texttt{true}$, and that for process 2 becomes $I_A(2) = \texttt{c}_2 = \texttt{false} \wedge \texttt{z}_2 = \texttt{true}$. The transition relation N_C can also be abstracted as N_A with respect to the predicates above. For example, the first transition of process 1 can be tracked by the guarded command $\texttt{z}_1 \longrightarrow \texttt{z}_1 := \texttt{false}; \texttt{p} := \texttt{false}$ which can be expressed by the transition relation $z_1 \wedge \neg z_1' \wedge \neg p' \wedge c_1' = c_1 \wedge c_2' = c_2 \wedge z_2' = z_2$, where primed variables represent the value of the variable in the end state of the transition. Once the abstract finite-state transition system has been constructed, it can be model checked. Such abstractions can also be defined to reduce a finite-state system to a smaller one in order to make the search space more manageable.

More generally, any concrete formula P can be abstracted by a formula $\alpha(P)$. For an abstract formula $\widehat{P}$, let $\gamma(\widehat{P})$ be the concrete formula that results from substituting the concrete predicate for the abstract one. For example, the abstract formula $z_1 \wedge \neg z_1' \wedge \neg p' \wedge c_1' = c_1 \wedge c_2' = c_2 \wedge z_2' = z_2$, which corresponds to the abstract transition above and is abbreviated as $\widehat{P}$, can be concretized by substituting the predicates corresponding to the Boolean variables, to get

$$\gamma(\widehat{P}) = \begin{array}{l} \texttt{ticket}[1] = 0 \\ \wedge\ \texttt{ticket}'[1] \neq 0 \\ \wedge\ \texttt{ticket}[1] \not\leq \texttt{ticket}[2] \\ \wedge\ \texttt{critical}'[1] = \texttt{critical}[1] \\ \wedge\ \texttt{critical}'[2] = \texttt{critical}[2] \\ \wedge\ (\texttt{ticket}'[2] = 0) \iff (\texttt{ticket}[2] = 0) \end{array}$$

We can then check that $\vdash \gamma(\widehat{P}) \Rightarrow P$, where P is the formula corresponding to the concrete transition. We can use deduction to construct $\alpha(P)$ from P such that $\vdash \gamma(\alpha(P)) \Rightarrow P$. This can be done by using an SMT solver with an AllSMT capability for constructing all solutions to the formula $\exists \overline{x}.P \wedge \bigwedge_{i=0}^{n} p_i = P_i$, where $\overline{x}$ is a sequence of concrete variables and p_i is the Boolean variable abstracting the concrete predicate P_i [47]. Then $\alpha(P)$ can be defined as $\neg AllSMT(\exists \overline{x}.\neg P \wedge \bigwedge_{i=0}^{n} p_i = P_i)$. We can then argue that $\vdash \gamma(\alpha(P)) \Rightarrow P$, since no truth assignment for $\alpha(P)$ can be extended to a truth assignment for $\neg P$ satisfying the abstraction map $\bigwedge_{i=0}^{n} p_i = P_i$. Hence, it must be the case that $\gamma(\alpha(P))$, which is equivalent to $\alpha(P) \wedge \bigwedge_{i=0}^{n} p_i = P_i$, entails P.

A weaker abstraction can be constructed by using an abstract lattice consisting of *monomials*, i.e., conjunctions of literals, instead of the Boolean lattice of formulas [6, 75]. As with abstract interpretation, we can also compute the abstract transfer function for each individual Boolean variable. *Data abstraction*, where a datatype is partitioned into a finite set of regions, as for example, the partitioning of the integers into positive numbers, negative numbers, and zero, can also be viewed as a form of predicate abstraction.

Predicate abstraction builds an abstract transition system that over-approximates the concrete one. Each abstract state a represents a set of concrete states

$\{c|c \models \gamma(a)\}$. The abstract transition system is defined by an $\exists\exists$ abstraction such that there is a transition $N_A(a, a')$ between two abstract states a and a' whenever there exist concrete states c and c' such that $c \models \gamma(a)$, $c' \models \gamma(a')$, and $N_C(c, c')$ holds. Thus, if the set of abstract reachable states $\mu X.I \sqcup N[X]$ contains a state a' that is reachable from an abstract initial state a, then there is a path $\langle a_0, \ldots, a_n\rangle$ with $a = a_0$ and $a' = a_n$. Since we have an $\exists\exists$ abstraction, for each adjacent pair $\langle a_i, a_{i+1}\rangle$ with $0 \leq i < n$, there is a pair of concrete states $\langle c, c'\rangle$ such that $N(c, c')$ holds. However, this does not imply the existence of a concrete path of the form $\langle c_0, \ldots, c_n\rangle$ such that $c_i \models \gamma(a_i)$ since it is possible that for some a_{i+1}, the set of states $\{c'|\exists c.c \models \gamma(a_i), c' \models \gamma(a_{i+1}), N(c, c')\}$ has an empty intersection with the set of states $\{c'|\exists c''.c' \models \gamma(a_{i+1}), c'' \models \gamma(a_{i+2}), N(c', c'')\}$. The absence of a concrete path corresponding to an abstract path can be established by techniques similar to bounded model checking (see below). This means that abstraction must be refined so that the two sets of states: the successors of $c \in \gamma(a_i)$ and the predecessors of $c'' \in \gamma(a_{i+2})$, are distinguishable. Symbolic methods for computing strongest post-conditions and weakest preconditions can be used for computing predicates needed for an abstraction that prunes spurious abstract counterexamples. The technique of interpolation can also be used for this purpose. Abstraction-based approaches to model checking are covered in this Handbook [30] (Dams and Grumberg, Abstraction and Abstraction Refinement).

The technique of $\exists\exists$ abstraction can yield abstractions that over-approximate concrete behavior. In particular, every concrete computation can be simulated by an abstract one. Thus, for any μ-calculus property P with universal path quantification, the concrete transition system satisfies P if the abstract transition system satisfies the abstraction $\widehat{P}$. In particular, when proving an invariant P by showing that $\mu X.I \sqcup N[X] \Rightarrow P$, we can see that the latter formula has an existential path-quantified μ-calculus formula occurring negatively, which is hence a universal path-quantified formula. The dual problem is that of under-approximating a μ-calculus formula so as to preserve the validity of universal path-quantified μ-calculus formulas. This requires a $\forall\exists$ abstraction of the transition relation where we admit a transition between abstract states a and a' only when for every state $c \models \gamma(a)$, there is a state $c' \models \gamma(a')$ such that $N(c, c')$. This means that whenever there is an abstract path $\langle a_0, \ldots, a_n\rangle$ in the abstract transition system, there is a corresponding concrete path $\langle c_0, \ldots, c_n\rangle$ through the state sets $\langle \gamma(a_0), \ldots, \gamma(a_n)\rangle$.

20.3.4 Bounded Model Checking and k-Induction

Invariants are the most common class of properties about transition systems. An invariant is an assertion P that holds in every reachable state of the system, and the set of reachable states is indeed the strongest system invariant. Bounded model checking uses satisfiability to check a sequence of states of a transition system for states violating the invariant P. For a given transition system $\langle W, I, N\rangle$, the k-expansion $BM(I, N, k)$ of the system is a formula $I(s_0) \wedge \bigwedge_{i=0}^{k} N(s_i, s_{i+1})$ which characterizes the sequences $\langle s_0, \ldots, s_{k+1}\rangle$ representing an initial segment of $k + 1$ steps

in a computation. The violation of an invariant can be represented by the formula $BM(I, N, k) \wedge \bigvee_{i=0}^{k+1} \neg P(s_i)$. If the latter formula is satisfiable, then there is a reachable state where P does not hold.

As an exercise, the reader can verify the mutual exclusion property of Peterson's algorithm in Sect. 20.1. Recall that the transition system only had five Boolean variables, and hence at most 2^5 distinct states. Therefore every state should be reachable in at most 31 steps. In practice, the maximal loop-free paths are significantly smaller. The bound can be expanded by iterative deepening by incrementing the parameter k until a property violation is found. Since the bounded-model-checking proof obligation of $k+1$ steps includes the constraints for $k+1$ steps, the iterative deepening can retain the clauses learned in prior steps.

Bounded model checking can also be applied for checking liveness properties by specifying lasso-like counterexample traces that consist of a path from an initial state s_0 to a state s_k and then a nontrivial loop about s_k of length at least m, where a property P fails to hold. A counterexample to $\Diamond P$ can be constructed with a query of the form $I(s_0) \wedge \neg P(s_0) \wedge \bigwedge_{i=0}^{k-1}(\neg P(s_i) \wedge N(s_i, s_{i+1})) \wedge \bigwedge_{i=0}^{m}(\neg P(s_k) \wedge N(s_{i+k}, s_{i+k+1}) \wedge s_k = s_{m+k+1})$.

Satisfiability checking can also be applied to induction arguments. The customary technique of verifying an invariant P for a transition system $\langle W, I, N\rangle$ is by showing that P holds initially, i.e., $I(s_0) \wedge \neg P(s_0)$ is unsatisfiable, and that it is inductive, i.e., $P(s) \wedge N(s, s') \wedge \neg P(s')$ is unsatisfiable. As is well known, the proof obligations can fail not because the invariant is not valid, but because it is not inductive. The inductivity proof obligation can fail because of a counterexample where the state s is unreachable. In k-induction, the base case is identical to the k-step bounded-model-checking query, and the induction step checks the unsatisfiability of $\bigwedge_{i=0}^{k}(P(s_0) \wedge N(s_i, s_{i+1})) \wedge \neg P(s_{k+1})$. Since the state s_k must be reachable in no more than $k-1$ steps, this rules out a number of unreachable states. The usual technique for proving invariance is then just 1-induction. As with bounded model checking, the bound k can be iteratively increased. For example, the mutual exclusion property of the algorithm in Fig. 2 can be proved by 5-induction without the need for any invariant strengthening.

Bounded model checking and k-induction can also be applied to infinite-state systems by using SMT solvers instead of SAT solvers for checking satisfiability. This approach can be used to handle systems with timed behavior and datatypes such as integers, rationals, reals, sets, arrays, and recursively defined structures such as lists and trees. Iterative deepening with SMT solvers can exploit the retention of learned clauses as well as theory lemmas. Brown and Pike's verification [66] of the biphase mark and 8N1 communication protocols using the SAL model checker [57] illustrates the power of infinite-state k-induction. Bounded model checking is covered in this Handbook [10] (Biere and Kroening, SAT-Based Model Checking).

20.3.5 Symbolic Execution and Test Generation

Bounded model checking demonstrates the use of satisfiability solvers for generating counterexamples traces. The same technique can also be used for the symbolic

execution of paths through a program as a mechanism for generating test cases. In this case, the goal is to find a test case that reaches a particular control point in a program. For example, with the program in Fig. 1, we may want a test case where the program terminates with one unrolling of the loop. We wish to find an input such that the assertion $\texttt{max} = 0$ holds at termination. By depth-first search, we can find a path of the form

$$\begin{aligned} &\texttt{max}_0 = 0; \quad \texttt{i}_0 = 0; \quad (\texttt{i}_0 < \texttt{N}); \quad \neg\big(\texttt{a}[\texttt{i}_0] > \texttt{max}_0\big); \\ &\texttt{i}_1 = \texttt{i}_0 + 1; \quad \neg(\texttt{i}_1 < \texttt{N}); \quad (\texttt{max}_0 = 0). \end{aligned}$$

An SMT solver can be used to determine that $N = 0, a[0] = 0$ is a possible input. Unlike bounded model checking where the entire transition system is represented as a formula, symbolic execution generates the constraints corresponding to specific paths through the control-flow graph of the program. It can be combined with random testing to generate inputs that find inputs for paths not covered by the given inputs. One can also use concrete inputs in conjunction with symbolic execution to, for example, convert nonlinear constraints into linear ones to simplify constraint solving. This connection to test generation is explored in depth elsewhere in this Handbook [37] (Godefroid and Sen, Combining Model Checking and Testing).

20.3.6 Interpolation-Based Model Checking

We have already seen that whenever we have two sets of formulas that are jointly unsatisfiable, interpolation can be used to construct a formula in the shared language that captures the essence of the unsatisfiability. With bounded model checking, we see that the failed attempts to find a counterexample essentially amount to the BMC query being unsatisfiable. A BMC query has the form $I(s_0) \wedge \bigwedge_{i=0}^{k} N(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} \neg P(s_i)$. For any given $j < k$, we can split the query into the form $I(s_0) \wedge \bigwedge_{i=0}^{j} N(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{j} \neg P(s_i)$ and $\bigwedge_{i=j+1}^{k} N(s_i, s_{i+1}) \wedge \bigvee_{i=j+1}^{k+1} \neg P(s_i)$. The shared language consists of the variables of state $j + 1$ so that an interpolant yields an assertion about state $j + 1$ that is implied by the computation up to that point. Specifically, if we let j be 0, then we get an assertion $Q(s_1)$ that is implied by $I(s_0) \wedge N(s_0, s_1) \wedge \neg P(s_0)$. This means that instead of increasing the bound, we replace the initialization constraint by $I^1(s_0) = I(s_0) \vee Q(s_0)$. If by progressively weakening the initialization predicate in this proof-directed manner, one reaches a fixpoint where $I^{j+1}(s_0) = I^j(s_0)$, then this predicate is an invariant which implies the desired property P.

Interpolation is covered in this Handbook [54] (McMillan, Interpolation and Model Checking). It can also be used for abstraction refinement in conjunction with predicate abstraction.

20.3.7 Conflict-Directed Reachability

We have already seen that inductive invariants are critical for deductive verification. The key to constructing inductive invariants is to find an over-approximation to the set of reachable states that is stronger than the desired property. Conflict-Directed Reachability (CDR) [16, 17] is a technique for constructing an inductive invariant Q for a transition system $M = \langle W, I, N\rangle$ that is stronger than the desired invariant P. We describe an abstract CDR algorithm below. For predicates S_1 and S_2, let $S_1 \sqsubseteq S_2$ hold when it is the case that $\forall s.S_1(s) \Rightarrow S_2(s)$. This is verified by checking that the formula $S_1(s) \wedge \neg S_2(s)$ is unsatisfiable. Let $S_1 \sqcap S_2$ ($S_1 \sqcup S_2$) represent the intersection (union) of predicates S_1 and S_2, and $\neg S$ represent the complement of S. The CDR algorithm constructs a sequence of state predicates $Q_0, \ldots, Q_n$, where the ith predicate Q_i over-approximates the set of states reachable in i or fewer steps, i.e., $\bigsqcup_{k=0}^{i} N^k[I] \sqsubseteq Q_i$, where $N[S]$ is the image of S under N. For a transition system $M = \langle W, I, N\rangle$, let $M[X] = I \sqcup N[X]$, and $N^{-1}[X]$ represent the pre-image of X under N. We have the following properties that are maintained for the sequence $Q_0, \ldots, Q_n$:

1. $Q_0 = I$.
2. $Q_i \sqsubseteq Q_j \sqcap P$ for $i < j \leq n$.
3. $M[Q_i] \sqsubseteq Q_{i+1}$, for $0 \leq i < n$.

The CDR algorithm also maintains for each i, $0 \leq i \leq n$, a set C_i of *symbolic counterexamples*, where each $R \in C_i$ is a state predicate such that

1. $R = \neg P$ and $i = n$, or there is an S in $C_{i'}$, $R \sqsubseteq N^{-1}[S]$, where $i' = n$ if $i = n$, and $i' = i + 1$, otherwise.
2. $Q_j \sqsubseteq \neg R$ for all $j < i$.
3. $R \sqcap Q_i$ is nonempty, i.e., $Q_i \not\sqsubseteq \neg R$.

Each R in C_i, for $0 \leq i \leq n$, is a *counterexample to inductivity* (CTI) that is part of a symbolic trace $R_1, \ldots, R_m$ with $R_1 = R$ and $R_m = \neg P$, where $R_i \sqsubseteq N^{-1}[R_{i+1}]$ for $0 < i < m$. Thus at any stage, the algorithm maintains a sequence $Q_0, \ldots, Q_n$ where each Q_i over-approximates the set of states reachable in i or fewer steps. If $Q_{i+1} \sqsubseteq Q_i$ for some i, then Q_i is an inductive invariant since $N[Q_i] \sqsubseteq Q_{i+1} \sqsubseteq Q_i$. The algorithm works by progressively strengthening the state predicates Q_i by identifying and eliminating counterexamples to inductivity. If we succeed in extending the symbolic counterexample back to the initial predicate Q_0, then we have a concrete counterexample trace leading from an initial state to one where $\neg P$ holds. Otherwise, we can strengthen the predicates Q_i to eliminate the symbolic counterexamples. This process is continued until we find a concrete counterexample trace or an inductive invariant Q strengthening P.

Initially, $n = 0$ and the sequence consists of just one predicate Q_0. We assume that $Q_0 \sqsubseteq P$ since we have an immediate counterexample otherwise. In each subsequent step, the sequence $Q_0, \ldots, Q_n$ is progressively strengthened to eliminate the counterexamples to inductivity in one of the following ways:

1. **Fail:** If C_0 is nonempty, $\neg P$ is reachable.
2. **Succeed:** If $Q_i = Q_{i+1}$ for some $i < n$, we have an inductive weakening of P.
3. **Extend:** If C_i is empty for each $i \leq n$, add Q_{n+1} such that $M[Q_n] \sqsubseteq Q_{n+1}$ and $C_{n+1} = \emptyset$, if $Q_{n+1} \sqsubseteq P$, and $C_{n+1} = \{\neg P\}$, otherwise.
4. **Refine:** Check $N[Q_i] \sqsubseteq \neg R$ for some R in C_{i+1}, where C_j is empty, for $j \leq i$:
   a. **Strengthen:** If the query succeeds, find an $\widehat{R}$ weakening R that is relatively inductive: $M[Q_i \sqcap \neg\widehat{R}] \sqsubseteq \neg\widehat{R}$: conjoin $\neg\widehat{R}$ to each Q_j for $1 \leq j \leq i+1$, move any $S \in C_{i+1}$ such that $Q_{i+1} \sqsubseteq \neg S$ (including R) to C_{i+2} if $i+1 < n$.
   b. **Reverse:** If the query fails with counterexample s, weaken s to S such that $S \sqsubseteq N^{-1}[R]$ and add S to C_i.
5. **Propagate:** Whenever Q_i is strengthened, strengthen Q_{i+1} with Q where $M[Q_i] \sqsubseteq Q$, move any $R \in C_{i+1}$ such that $Q_{i+1} \sqsubseteq \neg R$ to C_{i+2} if $i+1 < n$.

Each step of the algorithm either fails with a counterexample trace (**Fail**), finds an inductive invariant (**Succeed**), extends the sequence when there are no counterexamples (**Extend**), strengthens some Q_i (**Strengthen**) followed by a sequence of propagation steps (**Propagate**), or adds a new counterexample to some C_i (**Reverse**). The reader should verify that the rules preserve the invariants on the sequences $Q_0, \ldots, Q_n$ and $C_0, \ldots, C_n$. From these invariants, one can conclude that the algorithm either returns a valid counterexample showing a path from I to $\neg P$ where each step satisfies N, or it returns an inductive invariant Q entailing P. The abstract algorithm converges if we can place an upper bound on the number of times these steps can be executed, i.e., repeated strengthening of a state predicate, the length of a monotonically weakening sequence $Q_0, \ldots, Q_n$, and the number of distinct symbolic counterexamples. The termination therefore depends on the specific concrete instantiation of the algorithm. The individual steps of the algorithm can be executed through queries to a SAT or SMT solver with the ability to find unsatisfiable cores or relevant assumptions, and minimal satisfying partial assignments.

The abstract CDR algorithm can be instantiated for finite-state systems. For a transition system where the state variables $b_1, \ldots, b_m$ are Boolean, each Q_i can be represented as a set of clauses in these variables, and $Q_{i+1} \sqsubseteq Q_i$, for $0 < i < n$. Here, Q_i represents the state predicate mapping the state s consisting of an assignment of truth values to $b_1, \ldots, b_m$ to the truth value of the conjunction of the clauses in Q_i. The transition relation N is a Boolean formula in the variables $b_1, \ldots, b_m, b'_1, \ldots, b'_m$. The primed version K' of a clause K is the result of replacing each b_i in K by b'_i for $1 \leq i \leq m$. The image $N[Q_i]$ consists of the clauses K in Q_i such that $Q_i \wedge N \wedge \neg K'$ is unsatisfiable. Each counterexample R is a cube, a conjunction of literals, in the variables $b_1, \ldots, b_m$. The CDR algorithm converges in this case because we cannot have a monotonically weakening sequence of predicates without finding a duplicate pair of adjacent predicates in the sequence, namely an inductive invariant.

20.4 Conclusions

Deduction and model checking are complementary approaches to verification. The former relies on assertions and program structure to construct proofs, whereas the latter is based on actually computing exact or approximate fixpoints. Model checking is effective at computing small properties of large systems, whereas deduction is needed for establishing correctness properties that require nontrivial mathematical reasoning. There is considerable synergy in terms of the reasoning tools: SAT and SMT solving, interpolants, quantifier elimination procedures, and decision procedures. We have outlined the foundations of deductive verification and explored only a few of the rich range of connections to model checking. It should be clear from the examples shown here that deduction is already a key element of many modern verification algorithms. In the algorithms that we have covered, deductive methods are used for symbolic execution, bounded state exploration, predicate abstraction, abstract transfer function construction, abstraction refinement, interpolant generation, model construction, proof construction, and test case generation. It is reasonable to expect that many significant advances in automated verification will be sparked by the complementarity and synergy between deduction and model checking.

Acknowledgements This research was supported by NSF Grants CSR-EHCS(CPS)-0834810, and CNS-0917375, and by NASA Cooperative Agreements NNA10DE73C and NNA13AC55C.

The chapter has been significantly improved by the valuable feedback from the anonymous reviewers, by the thorough, insightful, and constructive corrections and suggestions from Sriram Sankaranarayanan, Prashanth Mundkur, and Tomer Kotek, and insightful advice on the CDR algorithm from Aaron Bradley, Bruno Dutertre, and Dejan Jovanović. The author is deeply grateful to them, but takes full responsibility for any remaining mistakes or deficiencies.

References

1. Abdulla, P.A., Sistla, A.P., Talupur, M.: Model checking parameterized systems. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
2. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. **126**(2), 183–235 (1994)
3. Andrews, P.B.: An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof. Academic Press, Cambridge (1986)
4. Apt, K.R.: Ten years of Hoare's logic: a survey—Part 1. Trans. Program. Lang. Syst. **3**(4), 431–483 (1981)
5. Arora, S., Barak, B.: Computational Complexity: A Modern Approach. Cambridge University Press, Cambridge (2009)
6. Ball, T., Majumdar, R., Millstein, T., Rajamani, S.: Automatic predicate abstraction of C programs. In: Conf. on Programming Language Design and Implementation (PLDI), pp. 203–313. ACM, New York (2001)
7. Barrett, C.W., Tinelli, C.: Satisfiability modulo theories. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
8. Barwise, J.: First-order logic. In: Barwise, J. (ed.) Handbook of Mathematical Logic, Studies in Logic and the Foundations of Mathematics, vol. 90, pp. 5–46. North-Holland, Amsterdam (1978)

9. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Springer, Heidelberg (2004). Coq home page: http://coq.inria.fr/
10. Biere, A., Kroening, D.: SAT-based model checking. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
11. Blackburn, P., de Rijke, M., Venema, Y.: Modal Logic. Cambridge University Press, Cambridge (2002)
12. Bloem, R., Chatterjee, K., Jobstmann, B.: Graph games and reactive synthesis. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
13. Börger, E., Grädel, E., Gurevich, Y.: The Classical Decision Problem. Perspectives in Mathematical Logic. Springer, Heidelberg (1997)
14. Bouyer, P., Fahrenberg, U., Larsen, K.G., Markey, N., Ouaknine, J., Worrell, J.: Model checking real-time systems. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
15. Bradfield, J., Walukiewicz, I.: The mu-calculus and model checking. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
16. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D.A. (eds.) Intl. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI). LNCS, vol. 6538, pp. 70–87. Springer, Heidelberg (2011)
17. Bradley, A.R.: Understanding IC3. In: Cimatti, A., Sebastiani, R. (eds.) Theory and Applications of Satisfiability Testing (SAT). LNCS, vol. 7317, pp. 1–14. Springer, Heidelberg (2012)
18. Bradley, A.R., Manna, Z.: The Calculus of Computation: Decision Procedures with Applications to Verification. Springer, Heidelberg (2007)
19. Bryant, R.E.: Binary decision diagrams. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
20. Buss, S.R.: The Boolean formula value problem is in ALOGTIME. In: ACM Symposium on Theory of Computing (STOC), pp. 123–131. ACM, New York (1987)
21. Chaki, S., Gurfinkel, A.: BDD-based symbolic model checking. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
22. Chandy, K.M., Misra, J.: Parallel Program Design: A Foundation. Addison-Wesley, Reading (1988)
23. Church, A.: A formulation of the simple theory of types. J. Symb. Log. **5**, 56–68 (1940)
24. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
25. Constable, R.L., Allen, S.F., Bromley, H.M., Cleaveland, W.R., Cremer, J.F., Harper, R.W., Howe, D.J., Knoblock, T.B., Mendler, N.P., Panangaden, P., Sasaki, J.T., Smith, S.F.: Implementing Mathematics with the Nuprl Proof Development System. Prentice Hall, New York (1986). Nuprl home page: http://www.nuprl.org
26. Cook, S.A.: Soundness and completeness of an axiom system for program verification. SIAM J. Comput. **7**(1), 70–90 (1978)
27. Corbett, J.C., Dwyer, M.B., Hatcliff, J., Laubach, S., Păsăreanu, C.S., Robby, Zheng, H.: Bandera: extracting finite-state models from Java source code. In: International Conference on Software Engineering, pp. 439–448 (2000)
28. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: ACM Symposium on Principles of Programming Languages, pp. 238–252. ACM, New York (1977)
29. Craig, W.: Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. J. Symb. Log. **22**(3), 269–285 (1957)
30. Dams, D., Grumberg, O.: Abstraction and abstraction refinement. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)

31. Doyen, L., Frehse, G., Pappas, G.J., Platzer, A.: Verification of hybrid systems. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
32. Emerson, E.A.: Temporal and modal logics. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science Volume B: Formal Models and Semantics, pp. 997–1072. MIT Press/Elsevier, Cambridge/Amsterdam (1990)
33. Emerson, E.A., Clarke, E.M.: Using branching time temporal logic to synthesize synchronization skeletons. Sci. Comput. Program. **2**(3), 241–266 (1982)
34. Floyd, R.W.: Assigning meanings to programs. In: Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics, vol. XIX, pp. 19–32. AMS, Providence (1967)
35. Ganzinger, H., Rueß, H., Shankar, N.: Modularity and refinement in inference systems. Tech. Rep. CSL-SRI-04-02, SRI International, Computer Science Laboratory (2004). Revised, August 2004
36. Giannakopoulou, D., Namjoshi, K.S., Păsăreanu, C.S.: Compositional reasoning. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
37. Godefroid, P., Sen, K.: Combining model checking and testing. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
38. Gordon, M.J.C., Melham, T.F. (eds.): Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic. Cambridge University Press, Cambridge (1993). HOL home page: http://www.cl.cam.ac.uk/Research/HVG/HOL/
39. Harel, D.: First Order Dynamic Logic. LNCS, vol. 68. Springer, Heidelberg (1979)
40. Hennessy, M., Milner, R.: Algebraic laws for nondeterminism and concurrency. J. ACM **32**(1), 137–161 (1985)
41. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–583 (1969)
42. Holzmann, G.: Explicit-state model checking. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
43. Immerman, N.: Descriptive Complexity. Springer, Heidelberg (1999)
44. Jhala, R., Podelski, A., Rybalchenko, A.: Predicate abstraction for program verification. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
45. Jones, C.B.: The early search for tractable ways of reasoning about programs. IEEE Ann. Hist. Comput. **25**(2), 26–49 (2003)
46. Kaufmann, M., Manolios, P., Moore, J.S.: Computer-Aided Reasoning: An Approach. Advances in Formal Methods, vol. 3. Kluwer Academic, Norwell (2000)
47. Lahiri, S., Nieuwenhuis, R., Oliveras, A.: SMT techniques for predicate abstraction. In: Computer-Aided Verification, CAV. LNCS, vol. 4144, pp. 424–437. Springer, Heidelberg (2006)
48. Leivant, D.: Higher order logic. In: Gabbay, D.M., Hogger, C.J., Robinson, J.A. (eds.) Handbook of Logic in Artificial Intelligence and Logic Programming, Volume 2: Deduction Methodologies, pp. 229–321. Clarendon, Oxford (1994)
49. Li, Y., Albarghouthi, A., Kincaid, Z., Gurfinkel, A., Chechik, M.: Symbolic optimization with SMT solvers. In: Jagannathan, S., Sewell, P. (eds.) ACM Symposium on Principles of Programming Languages, pp. 607–618. ACM, New York (2014)
50. Majumdar, R., Raskin, J.F.: Symbolic model checking in non-Boolean domains. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
51. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems, Volume 1: Specification. Springer, Heidelberg (1992)
52. Marques-Silva, J., Malik, S.: Propositional SAT solving. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)

53. McCarthy, J.: A basis for a mathematical theory of computation. In: Braffort, P., Hershberg, D. (eds.) Computer Programming and Formal Systems. North-Holland, Amsterdam (1963)
54. McMillan, K.L.: Interpolation and model checking. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
55. Mints, G.E.: A short introduction to modal logic. No. 30 in CSLI lecture notes. Center for the Study of Language and Information (1992)
56. Morris, F.L., Jones, C.B.: An early program proof by Alan Turing. IEEE Ann. Hist. Comput. **6**(2), 139–143 (1984)
57. de Moura, L., Owre, S., Rueß, H., Rushby, J., Shankar, N., Sorea, M., Tiwari, A.: SAL 2. In: Computer-Aided Verification, CAV. LNCS, pp. 496–500. Springer, Heidelberg (2004). SAL home page: http://sal.csl.sri.com/
58. Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. Trans. Program. Lang. Syst. **1**(2), 245–257 (1979)
59. von Neumann, J.: John von Neumann, Collected Works, vol. V. Pergamon, Oxford (1961)
60. Neumann, J.v., Goldstine, H.H.: Planning and Coding of Problems for an Electronic Computing Instrument. Institute for Advanced Study, Princeton (1948)
61. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer, Heidelberg (2001)
62. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. Springer, Heidelberg (2002). Isabelle home page: http://isabelle.in.tum.de/
63. Owre, S., Rushby, J., Shankar, N., von Henke, F.: Formal verification for fault-tolerant architectures: prolegomena to the design of PVS. Trans. Softw. Eng. **21**(2), 107–125 (1995). PVS home page: http://pvs.csl.sri.com
64. Park, D.: Finiteness is mu-ineffable. Theor. Comput. Sci. **3**(2), 173–181 (1976)
65. Peterson, G.L.: Myths about the mutual exclusion problem. Inf. Process. Lett. **12**(3), 115–116 (1981)
66. Pike, L., Brown, G.M.: Easy parameterized verification of biphase mark and 8N1 decoders. In: Hermanns, H., Palsberg, J. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 3920, pp. 58–72. Springer, Heidelberg (2006)
67. Piterman, N., Pnueli, A.: Temporal logic and fair discrete systems. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
68. Pratt, V.R.: Semantical considerations on Floyd-Hoare logic. In: Annual Symposium on Foundations of Computer Science, pp. 109–121 (1976)
69. Queille, J.P., Sifakis, J.: Specification and verification of concurrent systems in CESAR. In: Proceedings of the 5th International Symposium on Programming. LNCS, vol. 137, pp. 337–351. Springer, Heidelberg (1982)
70. Rajan, S., Shankar, N., Srivas, M.: An integration of model-checking with automated proof checking. In: Wolper, P. (ed.) Computer-Aided Verification, CAV. LNCS, vol. 939, pp. 84–97. Springer, Heidelberg (1995)
71. Reps, T., Sagiv, M., Yorsh, G.: Symbolic implementation of the best transformer. In: Steffen, B., Levi, G. (eds.) Intl. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI). LNCS, vol. 2937, pp. 252–266. Springer, Heidelberg (2004)
72. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: Annual IEEE Symposium on Logic in Computer Science, pp. 55–74 (2002)
73. Rondon, P.M., Kawaguchi, M., Jhala, R.: Liquid types. In: Gupta, R., Amarasinghe, S.P. (eds.) Conf. on Programming Language Design and Implementation (PLDI), pp. 159–169. ACM, New York (2008). doi:10.1145/1375581.1375602
74. Rushby, J.M., von Henke, F., Owre, S.: An introduction to formal specification and verification using EHDM. Tech. Rep. SRI-CSL-91-2, Computer Science Laboratory, SRI International (1991)
75. Saïdi, H., Graf, S.: Construction of abstract state graphs with PVS. In: Computer-Aided Verification, CAV. LNCS, pp. 72–83. Springer, Heidelberg (1997)

76. Saïdi, H., Shankar, N.: Abstract and model check while you prove. In: Computer-Aided Verification, CAV. LNCS, pp. 443–454. Springer, Heidelberg (1999)
77. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Scalable analysis of linear systems using mathematical programming. In: Cousot, R. (ed.) Intl. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI). LNCS, vol. 3385, pp. 25–41. Springer, Heidelberg (2005)
78. Shankar, N.: Automated deduction for verification. ACM Comput. Surv. **41**(4), 20:1–20:56 (2009). doi:10.1145/1592434.1592437
79. Turing, A.M.: Checking a large routine. In: Ince, D.C. (ed.) Collected Works of A.M. Turing: Mechanical Intelligence, pp. 129–131. North-Holland, Amsterdam (1992). Originally presented at EDSAC Inaugural Conference on High Speed Automatic Calculating Machines, 24 June, 1949

Chapter 21
Model Checking Parameterized Systems

Parosh Aziz Abdulla, A. Prasad Sistla, and Muralidhar Talupur

Abstract We consider the model-checking problem for a particular class of *parameterized systems*: systems that consist of arbitrary numbers of components. The task is to show correctness regardless of the number of components. The term *parameterized* refers to the fact that the size of the system is a parameter of the verification problem. Examples of parameterized systems include mutual exclusion algorithms, bus protocols, networking protocols, cache coherence protocols, web services, and sensor networks. In this chapter, we will give four examples of techniques that have been used (among many others) for the verification of parameterized systems.

21.1 Introduction

The behavior of many types of computer systems can be described using one or more *parameters*, each of which varies over a specified range. The *parameterized verification problem* is to prove or refute that some specification is true for all values of the parameters. The parameters may relate to the size or topology of a network, to data types over which a system is constructed, to initial values of variables, etc. In this chapter, we concentrate on a particular class of parameterized systems, namely systems consisting of an arbitrary number of components (processes). The size of the system becomes (implicitly) a parameter of the verification problem, and parameterized verification amounts to proving correctness regardless of the number of processes. For instance, the specification of a mutual exclusion protocol may be parameterized by the number of processes participating in a given session of the protocol. In such a case, it is important to verify correctness independently of the number of participants in a particular session. There are numerous applications where

P.A. Abdulla (✉)
Uppsala University, Uppsala, Sweden
e-mail: parosh.abdulla@it.uu.se

A.P. Sistla
University of Illinois, Chicago, IL, USA

M. Talupur
Intel's Strategic CAD Labs, Hillsboro, OR, USA

E.M. Clarke et al. (eds.), *Handbook of Model Checking*,
DOI 10.1007/978-3-319-10575-8_21

parameterized systems appear naturally, such as mutual exclusion algorithms, bus protocols, network protocols, cache coherence protocols, web services, and sensor networks. Sensor networks typically consist of thousands of sensors, web services may attract millions of users, and network protocols may involve thousands or millions of nodes. More futuristic examples include the analysis of biological systems (e.g., organs with millions of cells) and social networks (with millions of users). All these examples show that there is a compelling need to extend verification from individual small instances to the parameterized case.

The nature of the problems that arise in parameterized verification and their solutions is determined by three factors:

- *Components*. The processes may be finite-state or infinite-state. Even in the case of finite-state processes, the state space of the system is unbounded. This is true since the state space contains all states we get as we vary the parameter (size of the system). In fact, we are dealing with an infinite family of systems (one for each possible size), and therefore the total state space (the union of state spaces of all members of the family) is infinite in size. The individual processes may also be infinite-state since they may operate on variables ranging over infinite domains (e.g., the natural numbers). In such a case we get a state space that is infinite in two dimensions.
- *Topology*. On the one hand, the system may consist of a set of processes without any structure. On the other hand, the system topology may have a certain pattern. For instance, the processes may be organized as a linear array. Then, a process may refer to its left/right neighbors, or to all the processes to its left/right. The processes may also be organized in a ring, tree, or a general graph.
- *Communication Primitives.* A simple form of communication is when two processes perform a *rendezvous* which involves both processes changing state simultaneously. Another form of communication is through *shared variables* that can be read from and written to by all/some processes in the system. We may have *broadcast transitions* where an arbitrary number of processes change state simultaneously. Furthermore, the transitions of a process may be conditioned by *global conditions*. An example of a (universal) global condition, in a system with linear topology, is that "all processes to the left of a given process i should satisfy a property ϕ". In this case, process i is allowed to perform the transition only in the case where all processes with indices $j < i$ satisfy ϕ.

An extensive research effort has been devoted to the verification of parameterized systems in recent years (see Sect. 21.6 for a survey). It is known that the parameterized verification problem is undecidable even in the case where each process is finite-state. In [8] it is shown that the following problem is not even semi-decidable (i.e., recursively enumerable) in general: Given a system $S(n)$ consisting of n finite-state processes and a specification $\phi(n)$ parameterized by n, check whether $S(n)$ satisfies $\phi(n)$ for all n. In [64], it is shown that the problem remains undecidable even if the system consists of a unidirectional ring of identical finite-state processes. In view of the undecidability of parameterized verification in general, research in the area has been conducted along three main lines:

- Identifying sub-classes for which the problem is decidable. This is often achieved by capturing the system's behavior using decidable models such as Petri nets.
- Designing algorithms that are not guaranteed to terminate in general, but that are augmented with *acceleration techniques*. Such techniques make the fixed-point computation underlying the verification algorithm terminate more often in practice.
- Using *over-approximation*. Here we apply *abstraction* techniques so that the verification problem for the abstract system is simpler than that of the original system, and such that correctness of the former implies correctness of the latter. One can also employ techniques based on *under-approximation*. Such methods, although not sound for verification, can aid in debugging as they may lead to more efficient error detection.

Due to space limitations, we will mainly concentrate on methods for the verification of safety properties, occasionally mentioning how liveness properties can also be handled. Furthermore, we will only describe the main ideas behind three existing techniques (among many others that have been introduced in the literature). More precisely, in Sect. 21.2 we consider systems in which the components are identical anonymous finite-state processes that communicate through shared variables. We show how the behavior of such a system can be modeled by a Petri net [38] which can then be analyzed using the theory of *well quasi-ordered transition systems* [2, 37]. In Sect. 21.3 we introduce *regular model checking* [8] which uses the theory of regular languages as a uniform framework for parameterized system verification. We give an example of a simple acceleration technique that can be defined within the framework. In Sect. 21.4, we introduce *monotonic abstraction*, a technique that aims at increasing the efficiency of regular model checking by computing an over-approximation. In Sect. 21.5 we present an abstraction-and-compositional-reasoning-based technique that has been successfully applied to industrial-strength protocols. Finally, we give an overview of related work in Sect. 21.6.

21.2 Petri Nets

In this section, we consider parameterized systems consisting of an unbounded number of identical finite-state processes that communicate through a finite set of shared variables each ranging over a finite domain. We describe how the behavior of such a parameterized system can be captured by the classical model of Petri nets. We show how safety properties of the system can be analyzed by using backward reachability analysis on Petri nets. Finally, we briefly outline how liveness properties can be analyzed.

21.2.1 Simple Protocol

To illustrate our ideas, we use a simple example of a mutual exclusion protocol (Fig. 1). The system consists of an arbitrary number of identical finite-state pro-

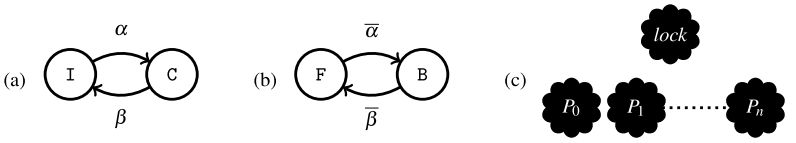


Fig. 1 (**a**) One process P_i in the simple protocol; (**b**) the lock; (**c**) a parameterized system consisting of an arbitrary number of processes

cesses competing for a global resource. The resource is guarded by a lock which is a shared variable among the processes. The system is supposed to satisfy *mutual exclusion*, i.e., at most one process may have access to the global resource at any given time. In each step of an execution of the system, one process, called the *active* process, performs a *local transition* changing its state. The rest of the processes, called the *passive* processes, do not change state. A process has two local states, namely `I` where the process is idle and `C` where the process is in its critical section. The state of the lock is `F` when it is free and `B` when it is busy. When a process wants to access its critical section, it must first acquire the lock. This can be done only if no other process has already acquired the lock. Intuitively, we can think of the shared-variable lock also as a process. The original processes communicate with the lock process in a rendezvous manner. Concretely, if a process is in state `I` (hence is capable of performing the event α) and the lock is in state `F` (hence is capable of performing the complementary event $\overline{\alpha}$), then they may move simultaneously changing to the states `C` and `B`, respectively (see Chap. 32 for the definition of complementary events). A similar behavior is induced by the events β and $\overline{\beta}$, which means that a process in the critical section may release the lock moving back to the idle state `I`. We require that the system never reaches a configuration where two or more processes are in the state `C`. Recall that we are interested in *parameterized verification*, i.e., verifying that this property is satisfied regardless of the number of competing processes.

21.2.2 Petri Nets

A Petri net $\mathcal{N}$ is a tuple (P, T, F), where P is a finite set of *places*, T is a finite set of *transitions*, and $F \subseteq (P \times T) \cup (T \times P)$ is the *flow relation*. For a place $p \in P$ and a transition $t \in T$, if $(p, t) \in F$ then p is said to be an *input* place of t; and if $(t, p) \in F$ then p is said to be an *output* place of t. We use $In(t) := \{p \mid (p, t) \in F\}$ and $Out(t) := \{p \mid (t, p) \in F\}$ to denote the sets of input places and output places of t, respectively.

Figure 2 shows an example of a Petri net with four places (drawn as circles), namely `I`, `F`, `B`, and `C`, and two transitions (drawn as rectangles), namely t_1 and t_2. The flow relation is represented by edges from places to transitions, and from transitions to places. For instance, the flow relation in the example includes the pairs $(\mathtt{F}, t_1)$ and $(t_2, \mathtt{I})$, i.e., `F` is an input place of t_1, and `I` is an output place of t_2.

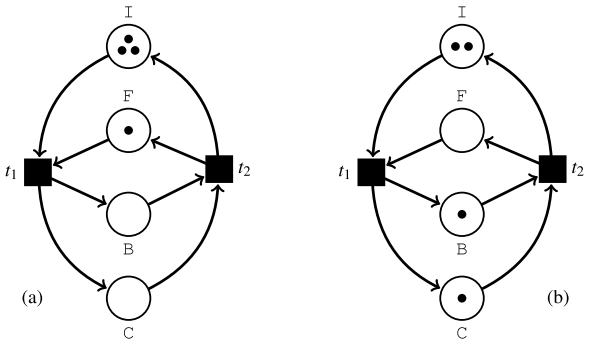


Fig. 2 (**a**) A simple Petri net; (**b**) the result of firing t_1

The transition system induced by a Petri net is defined by the set of *configurations* together with the *transition relation* defined on them. To give the definition, we recall some simple notations for *multisets*. We will write multisets (over some set A) as lists, so if $a, b \in A$ then $[a^3, b^2]$ represents a multiset M over A where $M(a) = 3$, $M(b) = 2$ and $M(x) = 0$ for $x \neq a, b$. For multisets M_1 and M_2, we write $M_1 \leq M_2$ if $M_1(a) \leq M_2(a)$ for all $a \in A$. We define the addition $M_1 + M_2$ to be the multiset M where $M(a) = M_1(a) + M_2(a)$, and (assuming $M_1 \leq M_2$) we define the subtraction $M_2 - M_1$ to be the multiset M where $M(a) = M_2(a) - M_1(a)$, for each $a \in A$. We define $M_1 \ominus M_2$ to be the multiset M where $M(a) = M_1(a) \ominus M_2(a)$ with $y \ominus x := \max(y - x, 0)$. A set U of multisets is said to be *upward closed* if whenever $M_1 \in U$ and $M_1 \leq M_2$ then $M_2 \in U$.

A *configuration* c of a Petri net[1] is a multiset over P. The configuration c defines the number of *tokens* in each place. Figure 2(a) shows a configuration with one token in place `F`, three tokens in place `I`, and no token in the places `B` and `C`. This configuration corresponds to the multiset $[\mathtt{F}, \mathtt{I}^3]$. For a transition t, we can view the set $In(t)$ of input places as a multiset over P where $In(t)(p) = 1$ if $p \in In(t)$, and $In(t)(p) = 0$ if $p \notin In(t)$. For instance, in Fig. 2, we have that $In(t_1) = [\mathtt{I}, \mathtt{F}]$. We can view the set $Out(t)$ of output places as a multiset in a similar manner. The operational semantics of a Petri net is defined through the notion of *firing*. More precisely, when a transition t is fired, a token is removed from each input place, and a token is added to each output place of t. The transition is fired only if each input place has at least one token. This defines a transition relation on the set of configurations. Formally, for a transition $t \in T$, we write $c_1 \xrightarrow{t} c_2$ to denote that $c_1 \geq In(t)$ and $c_2 = c_1 - In(t) + Out(t)$. We use $c_1 \longrightarrow c_2$ to denote that $c_1 \xrightarrow{t} c_2$ for some $t \in T$. For sets C_1, C_2 of configurations, we write $C_1 \longrightarrow C_2$ to denote that $c_1 \longrightarrow c_2$ for some $c_1 \in C_1$ and $c_2 \in C_2$. We define $\xrightarrow{*}$ as the reflexive transitive closure of $\longrightarrow$.

Suppose that we are given a parameterized system in which the processes are finite-state, and the processes communicate through a finite set of shared variables each ranging over a finite domain. We model the behavior of such a system using

[1] A configuration of a Petri net is often called a *marking* in the literature.

a Petri net. The idea is to *count* the number of processes in each local state. More precisely, we devote a place $\mathtt{q}$ in the Petri net to each process state q in the parameterized system. The number of tokens in the place $\mathtt{q}$ in the Petri net represents the number of processes in state q. Furthermore, for each shared variable x and value v in the domain of x, we have a place $\mathtt{v}$, where a token in $\mathtt{v}$ means that the value of x is v. For instance, we can describe the behavior of the parameterized version of the simple mutual exclusion protocol as the Petri net of Fig. 2. The numbers of tokens in places $\mathtt{I}$ and $\mathtt{C}$ represent the number of processes in their idle states and critical sections respectively. We encode the value of the lock using the places $\mathtt{F}$ and $\mathtt{B}$, where a token in the former means that the lock is free, and a token in the latter means that the lock is busy. Each transition of the Petri net corresponds to one of the processes performing a local transition: the transition t_1 corresponds to a process moving from $\mathtt{I}$ to $\mathtt{C}$ (thus decreasing the number of processes in state $\mathtt{I}$, increasing the number of processes in $\mathtt{C}$, and taking the lock); and the transition t_2 corresponds to a process moving from $\mathtt{C}$ to $\mathtt{I}$ (thus increasing the number of processes in state $\mathtt{I}$, decreasing the number of processes in $\mathtt{C}$, and releasing the lock).

21.2.3 Safety Properties

A safety property states that "nothing bad happens" during the execution of the system (see Chap. 2). Such a property can be formulated in terms of a set *Bad* of *bad configurations*. These are configurations that should not occur during the execution of the system (otherwise the system is considered to be violating the safety property). Thus, checking the safety property can be reduced to checking the reachability of the set of bad configurations.

For instance, in Fig. 2, the set *Bad* contains those configurations that violate mutual exclusion, i.e., configurations where at least two processes are in their critical sections. These configurations are of the form $[\mathtt{F}^{k_1}, \mathtt{B}^{k_2}, \mathtt{I}^{k_3}, \mathtt{C}^{k_4}]$ where $k_4 \geq 2$. The set of bad configurations is typically upward closed. This is the case in our example: whenever a configuration contains two processes in their critical sections then any larger configuration will also contain (at least) two processes in their critical sections.

The set C_{init} of *initial configurations* are those from which the execution of the system may start. In our example, the initial configurations are those where the lock is free and where all processes are idle. These configurations are of the form $[\mathtt{F}, \mathtt{I}^m]$ where $m \geq 1$. Examples of initial configurations are $[\mathtt{F}, \mathtt{I}^2]$ and $[\mathtt{F}, \mathtt{I}^5]$, corresponding to instances of the system with two and five processes respectively. Notice that there are infinitely many initial configurations (one for each possible size of the system).

Checking the safety property can be carried out by checking whether there exists a sequence of transitions leading from an initial configuration to a bad configuration, i.e., checking whether the set *Bad* is reachable (whether $C_{init} \xrightarrow{*} Bad$). Since the set *Bad* is upward closed, checking safety properties amounts to the problem of

checking the reachability of an upward-closed set (this problem is usually referred to as the *coverability problem* for Petri nets).

21.2.4 Backward Reachability Analysis

Two main methods for solving the coverability problem for Petri nets have been proposed in the literature. The first is based on *forward* reachability analysis (the Karp–Miller algorithm [43]), while the second is based on *backward* reachability analysis (by applying the theory of *well quasi-ordered programs* [2, 37]). In this chapter, we will concentrate on the latter since it is applicable to a larger class of (infinite-state) systems. In fact, we will see another application of this framework in Sect. 21.4.

We will introduce a backward reachability algorithm for solving the coverability problem. The algorithm relies on two properties of $\leq$. First, we notice that the transition relation $\longrightarrow$ is *monotonic* w.r.t. $\leq$. In other words, given configurations c_1, c_2, and c_3, if $c_1 \longrightarrow c_2$ and $c_1 \leq c_3$, then there exists a configuration c_4 such that $c_2 \leq c_4$ and $c_3 \longrightarrow c_4$. Second, according to Dickson's lemma [28], the relation $\leq$ is a *well quasi-ordering* (*wqo* for short), i.e., for each infinite sequence $c_0, c_1, c_2, \ldots$ of configurations there exist i and j such that $i < j$ and $c_i \leq c_j$.

The backward reachability algorithm manipulates sets of configurations that are upward closed. For a configuration c, we define $\widehat{c}$ to be its upward closure, i.e., $\widehat{c} = \{c' \mid c \leq c'\}$. Notice that $c' \leq c$ implies $\widehat{c} \subseteq \widehat{c'}$. For a set C, we define $\widehat{C} := \bigcup_{c \in C} \widehat{c}$. For an upward-closed set U, we define the *generator* of U to be the set of minimal elements of U, i.e., the set G such that: (i) $\widehat{G} = U$, i.e., U can be generated from G by taking the upward closure of G w.r.t. $\leq$; and (ii) $a \leq b$ implies $a = b$ for all $a, b \in G$, i.e., the set G is canonical in the sense that all its elements are incomparable w.r.t. $\leq$. We use $gen(U)$ to denote the set of generators of U (notice that this set is uniquely defined for a given U). The generators of the set of bad configurations are often known to us. In our example, the set is the singleton $gen(Bad) = \{[\mathbb{C}^2]\}$. Upward-closed sets enjoy two properties that make them useful in the backward algorithm:

- The set $gen(U)$ is finite. This is true since otherwise we would have an infinite set of incomparable elements, which contradicts the wqo property. This means that each upward-closed set U can be characterized by a *finite* set of configurations, namely its generator $gen(U)$. The set $gen(U)$ is a finite characterization of U in the sense that $U = \widehat{gen(U)}$.
- Consider an upward-closed set U of configurations. By monotonicity of $\longrightarrow$ it follows that the *predecessor set* $\{c \mid c \longrightarrow U\}$ is upward closed [2]. In other words, the set of configurations from which we can reach U through the firing of a single transition is upward closed (upward-closedness is preserved by firing transitions backwards). For a configuration c and a transition t, we define $Pre(t)(c) := gen(\{c' \mid c' \xrightarrow{t} \widehat{c}\})$, i.e., it is the generator of the set of configurations from which we can reach the upward closure of c through a single firing

Algorithm 1 Backward Reachability Algorithm

Input: • $\mathcal{N}$: Petri net
• C_{init}: set of initial configurations
• $gen(Bad)$: generator of the set of bad configurations
Output: Is Bad reachable in $\mathcal{N}$?

```
1: i ← 0
2: C_0 ← gen(Bad)
3: repeat
4:    C_{i+1} ← C_i ∪ {c | (c ∈ Pre(C_i)) ∧ (¬∃c' ∈ C_i . c' ≤ c)}
5:    i ← i + 1
6: until C_i = C_{i−1}
7: if C_init ∩ ĉ_i ≠ ∅ then
8:    return true
9: else
10:   return false
11: end if
```

of t. We define $Pre(c) := \bigcup_{t \in T} Pre(t)(c)$. For a set C of configurations, we define $Pre(C) := \bigcup_{c \in C} Pre(c)$. For each configuration c and transition t, it is easy to see that $Pre(t)(c)$ has the single configuration $(c \ominus Out(t)) + In(t)$.

Sets of bad configurations are almost always upward closed as explained above. Therefore, checking the safety property amounts to deciding reachability of an upward-closed set. Now, we are ready to define the backward reachability algorithm (Algorithm 1). The algorithm starts from the generators of the set of bad configurations, and tries to find a path backwards through the transition relation to the set of initial configurations. It performs a number of rounds where, during each round, it derives the predecessors of the current set of configurations. Furthermore, for each newly generated configuration c, it checks whether we have already encountered another configuration $c' \leq c$. In such a case, we know that $\widehat{c} \subseteq \widehat{c'}$ and hence we can safely discard c from the analysis without the loss of any information (we say that c is *subsumed* by c'). The algorithm terminates when all the newly generated configurations (those generated in the current iteration i) are subsumed by configurations that have been generated in previous iterations. In such a case, we know that $C_i = C_{i-1}$, and hence the upward closure $\widehat{C_i}$ is the set of configurations from which Bad is reachable. Hence, Bad is reachable if and only if C_{init} and $\widehat{C_i}$ have a non-empty intersection. The algorithm is guaranteed to terminate since $\leq$ is a well quasi-ordering (we refer to [2] for a formal proof of termination).

Let us apply the algorithm to our example. As mentioned, the set $gen(Bad)$ is the singleton $\{[\mathtt{C}^2]\}$. Therefore, the algorithm starts from the configuration $c_0 = [\mathtt{C}^2]$, and repeatedly computes predecessors by applying the function Pre. From the configuration c_0, we go backwards and derive the generator of the set of configurations from which we can fire a transition and reach a configuration in $Bad = \widehat{c_0}$. Transition t_1 gives the configuration $c_1 = [\mathtt{F}, \mathtt{I}, \mathtt{C}]$, since $\widehat{c_1}$ contains exactly those configurations from which we can fire t_1 and reach a configuration in $\widehat{c_0}$. Analogously, tran-

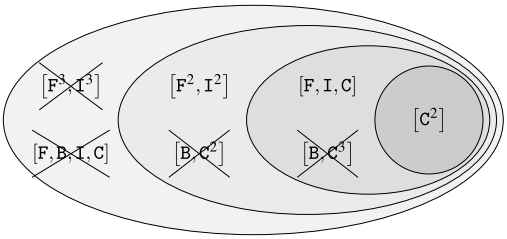


Fig. 3 Running the backward reachability algorithm on the example Petri net. Each ellipse contains the configurations generated during one iteration. The subsumed configurations are crossed out

sition t_2 gives the configuration $c_2 = [\mathtt{B}, \mathtt{C}^3]$, since $\widehat{c_2}$ contains exactly those configurations from which we can fire t_2 and reach a configuration in $\widehat{c_0}$. Since $c_0 \preceq c_2$, c_2 is subsumed by c_0 and c_2 will be discarded. Now, we repeat the procedure on c_1, and obtain the configurations $c_3 = [\mathtt{F}^2, \mathtt{I}^2]$ (via t_1), and $c_4 = [\mathtt{B}, \mathtt{C}^2]$ (via t_2), where c_4 is subsumed by c_0. Finally, from c_3 we obtain the configurations $c_5 = [\mathtt{F}^3, \mathtt{I}^3]$ (via t_1), and $c_6 = [\mathtt{F}, \mathtt{B}, \mathtt{I}, \mathtt{C}]$ (via t_2). The configurations c_5 and c_6 are subsumed by c_3 and c_1 respectively. The iteration terminates at this point since all the newly generated configurations were subsumed by existing ones, and hence there are no more new configurations to consider. The set $B = \{[\mathtt{C}^2], [\mathtt{F}, \mathtt{I}, \mathtt{C}], [\mathtt{F}^2, \mathtt{I}^2]\}$ is the generator of the set of configurations from which we can reach a bad configuration. The three elements of B are those configurations which are not discarded in the analysis (they were not subsumed by other configurations). To check whether *Bad* is reachable, we check the intersection $\widehat{B} \cap C_{init}$. Since the intersection is empty, we conclude that *Bad* is not reachable, and hence the safety property is satisfied by the system.

We summarize the properties that need to be satisfied by a transition system (in general) in order to be able to use the above algorithm (see also [2]). Suppose that we are given a set of configurations C, a transition relation $\longrightarrow$, a set C_{init} of initial configurations, an ordering $\preceq$ on C, and an upward-closed set *Bad* characterized by its generator $gen(Bad)$. The needed properties are the following:

1. $\longrightarrow$ is monotonic with respect to $\preceq$. This implies that the predecessor set of an upward-closed set of configurations is upward closed.
2. $\preceq$ is a wqo. We need this property for two reasons: to represent upward-closed sets with a finite set of configurations (the generator of the set); and to guarantee termination of the iterative computation in the reachability analysis.
3. For each c, we can compute the (finite) set $gen(\{c' \mid c' \longrightarrow \widehat{c}\})$.
4. For configurations c and c', we can check whether $c \preceq c'$. This property is needed so that we can check whether a configuration may be discarded in the algorithm.
5. We can check emptiness of the intersection $\widehat{B} \cap C_{init}$ for a finite set B of configurations. This is needed in line 7 of the backward reachability algorithm.

Notice that the transition system induced by any Petri net satisfies Properties 1–4.

21.2.5 Liveness Properties

In this section, we briefly discuss the problem of checking liveness properties for our class of parameterized systems. To simplify the presentation, we consider a slightly extended model, where we assume that we have a family of systems of processes consisting of a single process $\mathscr{C}$, called a control process, together with an arbitrary number of identical user processes whose definition is given by $\mathscr{U}$. A system in this model having n user processes is denoted by $\mathscr{C} \times \mathscr{U}^n$. We consider the family of systems given by $\{\mathscr{C} \times \mathscr{U}^n \mid n > 0\}$. The processes communicate through rendezvous. We will only consider verification of properties of executions of the control process. We assume that the correctness property is specified by an LTL formula f that refers to the states of the control process using atomic propositions. To check liveness properties, we need to reason about fair computations of the system of processes. We consider simple weak fairness, which states that if a process is enabled continuously from any point onwards then it is eventually executed after that point (see Chap. 2). Intuitively, a process is said to be enabled in a global state if it can either make a local transition or can communicate with another process. For the example of Fig. 1, process P_0 is enabled in a global state in which process P_0 is in state I and lock is in state F. A process is said to be executed in a computational step from one global state to another if it involves a transition of that process. We will check for correctness by checking that there does not exist a fair computation of a system of the form $\mathscr{C} \times \mathscr{U}^n$, for some $n > 0$, that satisfies $\neg f$. To do this, we first construct a Büchi automaton $\mathscr{A}_{\neg f}$ that accepts all those sequences of states of $\mathscr{C}$ that do not satisfy f, i.e., that satisfy $\neg f$ (see Chap. 4). The automaton $\mathscr{A}_{\neg f}$ accepts an input by going through at least one of a set of designated states, called *accepting* states, infinitely often. We construct another process $\mathscr{C}'$ which is a product of $\mathscr{C}$ and $\mathscr{A}_{\neg f}$. Essentially, $\mathscr{C}'$ behaves like $\mathscr{C}$ and at the same time simulates $\mathscr{A}_{\neg f}$ on its executions. Each state of $\mathscr{C}'$ is a pair of the form (s, q) where s is a state of $\mathscr{C}$ and q is a state of $\mathscr{A}_{\neg f}$. Such a state is called an accepting state if q is an accepting state of $\mathscr{A}_{\neg f}$.

Now, we consider a family of systems of the form $\mathscr{C}' \times \mathscr{U}^n$, $n > 0$ and check whether there is a fair computation of these systems where an accepting state of $\mathscr{C}'$ appears infinitely often. Such a computation does not exist iff the original family of systems satisfies the correctness property f. We construct a Petri net $\mathscr{N}$ that captures the computations of the family of systems of the form $\mathscr{C}' \times \mathscr{U}^n$, $n > 0$. As stated earlier, each place of $\mathscr{N}$ uniquely corresponds to a state of either $\mathscr{C}'$ or $\mathscr{U}$. A place in $\mathscr{N}$ is called an *accepting* place if it corresponds to an accepting state of $\mathscr{C}'$. A configuration of $\mathscr{N}$ is called an *accepting* configuration if there is at least one token in one of the accepting places.

The infinite executions of $\mathscr{N}$, which are infinite sequences of configurations resulting from firing transitions, correspond, as indicated earlier, to the computations of the above family of systems of processes. We define a notion of fairness of infinite executions of $\mathscr{N}$ which correspond to computations of the systems of the form $\mathscr{C}' \times \mathscr{U}^n$. The fairness of executions of $\mathscr{N}$ is defined with respect to places in the Petri net as opposed to fairness of system executions being defined with respect to

processes. (Our notion of fairness in Petri nets is different from liveness of Petri nets considered in the literature.) Roughly speaking, fairness of an execution of $\mathcal{N}$ is defined as follows. Say that a place p is *enabled* in a configuration c if at least one transition t, having p as one of its input places, can be fired from c. We say that an infinite execution e of $\mathcal{N}$ is *fair* if for every place p the following condition holds: if there exist infinite instances in e such that the place p is enabled in the configuration at that instance, then there are infinite instances in e when a transition having p as one of its input places is fired in e.

Now, we check for an infinite fair execution of $\mathcal{N}$ starting from the initial configuration that has accepting configurations occurring infinitely often. Existence of such an execution can be checked using the well-known *reachability* problem for Petri nets which consists of checking whether a particular configuration is reachable from the initial configuration. This problem is more difficult than that of checking whether any of an, upward-closed, set of configurations is reachable, which we considered earlier. The reachability problem for Petri nets is known to be decidable and hence checking correctness under fairness is decidable for our simple model of processes when the correctness property is given by an LTL formula on the executions of the control process. The details of the approach can be found in [38]. The above approach ignores deadlocked computations of the system of processes. However, it can be extended to consider deadlocked computations by extending them to infinite computations in a natural way. A similar approach can be used to check correctness of the executions of user processes.

21.3 Regular Model Checking

In this section, we consider parameterized systems with linear topologies in which the processes are finite-state and a process may communicate with its near (immediate) neighbors. Linear topologies are quite common in parameterized systems, where the position of a process may, for example, reflect its priority compared to the rest of the processes. Such a parameterized system induces undecidable verification problems (in fact, all non-trivial verification problems turn out to be undecidable). We introduce an example of an acceleration technique for the verification of such parameterized systems. We will formulate the technique in the framework of regular model checking.

21.3.1 Near-Neighbor Communication

We consider a class of parameterized systems consisting of finite-state processes organized in a linear array. The system allows a "near-neighbor" communication pattern in the sense that a given process may communicate with its left or right neighbor, and the two communicating processes may change state simultaneously.

As an example we will use a simple *token-passing protocol*. The system consists of an arbitrary (but finite) number of processes that are arranged as a linear sequence. Initially, the leftmost process owns the token. In each step, the process currently having the token communicates with its right neighbor (thus performing a near-neighbor communication) and passes the token to the right. An example of a safety property for this protocol is that at most one process will have the token at any point in the execution of the system.

Notice that the model introduced in Sect. 21.2 does not include near-neighbor communication. In fact, it is not hard to show that this simple extension leads to undecidability of safety properties. The undecidability proof can be carried out through a reduction from the halting problem for Turing machines in a similar way to the proof for ring-formed topologies [64]. Here, we only give a short outline of the proof, leaving the technical details to the reader. The intuition is that, given a Turing machine $\mathscr{T}$, a system of size n can simulate the computations of $\mathscr{T}$ that use at most n tape cells. Therefore, parameterized verification of the system amounts to the verification of $\mathscr{T}$ (which means that any non-trivial verification problem will be undecidable). Each cell is represented by one process whose state encodes the symbol stored in the cell. Furthermore, the state of the process contains a (Boolean) flag that indicates whether the head is currently pointing to the cell or not. If the value of the flag is *true* then the state of the process also stores the local state of $\mathscr{T}$. We can now use near-neighbor communication to simulate moving the head in $\mathscr{T}$, changing the current cell symbol, and changing the local state of $\mathscr{T}$.

In view of this undecidability proof, we will describe a verification method that is *not* guaranteed to terminate. The method is based on acceleration formulated in the framework of regular model checking.

21.3.2 Regular Model Checking

Regular Model Checking (RMC) [45] is a general paradigm for algorithmic verification of parameterized systems with linear or ring-shaped topologies. In fact, RMC is applicable to the verification of a much wider class of parameterized systems than the one with near-neighbor communication that we are dealing with here. In RMC, the system behavior is modeled in several steps. We start with a finite alphabet, where each symbol corresponds to the local state of a process. Configurations of the system are represented by words, where each position in the word describes the state of one process in the system. Sets of configurations are represented by finite automata (or equivalently by regular expressions). Finally, transitions are represented by finite automata operating on pairs of states, so called *finite-state transducers*. The idea of RMC is that we can exploit automata-theoretic algorithms for manipulating regular sets. Such algorithms have been successfully implemented, e.g., in the Mona [39] system.

Formally, let Σ be a finite alphabet of symbols. For a relation $R \subseteq \Sigma \times \Sigma$ and a set $A \subseteq \Sigma$, we define $A \circ R := \{b \mid \exists a.\ (a \in A) \wedge ((a, b) \in R)\}$. For relations

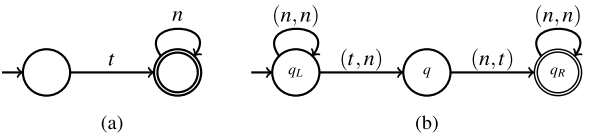


Fig. 4 (**a**) The set of initial configurations in the token-passing protocol. (**b**) The transducer describing the transition relation

$R, R' \subseteq \Sigma \times \Sigma$, we define the composition $R \circ R' := \{(a_1, a_2) \mid \exists b.\ ((a_1, b) \in R) \wedge ((b, a_2) \in R')\}$. For a natural number i, we define the relation R^i inductively by $R^0 = \{(a, a) \mid a \in \Sigma\}$ (i.e., it is the identity relation), and $R^{i+1} = R^i \circ R$. In other words, the relation R^i corresponds to i compositions of R. We define the reflexive transitive closure $R^* := \cup_{i \geq 0} R^i$, and the transitive closure $R^+ := \cup_{i \geq 1} R^i$.

A (finite-state) *transducer* T over Σ is a tuple (Q, q_{init}, Δ, F) where Q is a finite set of states, $q_{init} \in Q$ is the initial state, $\Delta \subseteq Q \times (\Sigma \times \Sigma) \times Q$ is the transition relation, and $F \subseteq Q$ is the set of accepting states. We use $q_1 \xrightarrow{(a,b)}_T q_2$ to denote that $(q_1, (a, b), q_2) \in \Delta$. A transducer is a finite-state automaton that accepts finite words over the alphabet $\Sigma \times \Sigma$, i.e., words of the form $(a_1, b_1)(a_2, b_2) \cdots (a_n, b_n)$. The language $L(T)$ of T is the set of words accepted by T. The transducer T induces a regular relation $R(T)$ on words (of identical lengths) over Σ. More precisely, for words $x = a_1 \cdots a_n$ and $y = b_1 \cdots b_n$ in Σ^*, we have $(x, y) \in R(T)$ if $(a_1, b_1) \cdots (a_n, b_n) \in L(T)$. We will use $R(T)$ to represent the transition relation on the configurations of the parameterized system (each of which is a word over Σ). To simplify the notation, we write $R^+(T)$ instead of $(R(T))^+$ to denote the transitive closure of $R(T)$. We use $R^*(T)$ and $R^i(T)$ in a similar manner.

We will describe how we model the token passing protocol in the framework of RMC (see Fig. 4). The alphabet is given by the set $\{t, n\}$, where t means that the process has the token, and n means that the process does not have the token. A configuration of the system is a word over Σ. For instance, the word $nntnn$ represents a configuration of the system with five processes where the third process has the token. The set of initial configurations is characterized by the regular expression tn^* (the finite-state automaton in Fig. 4(a)), indicating that the leftmost process has the token, and that this process is followed by an arbitrary number of processes, none of which has the token. The transition relation is represented by the transducer in Fig. 4(b). For instance, the transducer accepts the word $(n, n)(n, n)(t, n)(n, t)(n, n)$, representing the pair $(nntnn,\ nnntn)$ of configurations where the token is passed from the third to the fourth process. Notice that a pair of words (x, x') belongs to $R^i(T)$ if x can be rewritten to x' using i successive runs of T. For example, Fig. 5

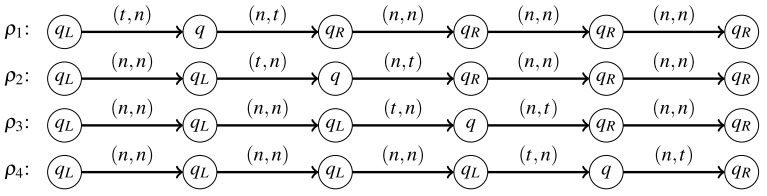
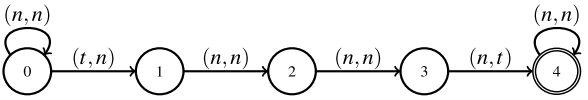


Fig. 5 Four runs of the transducer of Fig. 4

Fig. 6 The transducer T^3



shows four runs $\rho_1, \rho_2, \rho_3, \rho_4$ of the transducer T of Fig. 4(b). Each of these runs is of (the same) length 5. The runs relate the pairs $(tnnnn, ntnnn)$, $(ntnnn, nntnn)$, $(nntnn, nnntn)$, and $(nnntn, nnnnt)$, each of which belongs to $R(T)$. The pair $(tnnnn, nnnnt) \in R^4(T)$, as witnessed by the sequence of runs $\rho_1, \rho_2, \rho_3, \rho_4$.

21.3.3 Acceleration Techniques

A generic task in RMC is to compute a representation for the transitive closure of a transducer relation. Given a transducer T, we would like to construct a new transducer T^+ such that $R(T^+) = R^+(T)$, i.e., the relation characterized by T^+ is the transitive closure of the relation characterized by T. The transducer T^+ can then be used for computing the set of reachable configurations (when verifying safety properties), or to find loops (when verifying liveness properties). Below we describe how verification of safety properties can be carried out using the transitive closure (for liveness properties, the reader is referred to [6]).

Safety Properties

As mentioned in the previous section, a safety property can be verified by solving the *reachability* problem. Formulated in the RMC framework, the corresponding problem is the following: given a regular set of initial configurations I, a regular set of *bad configurations* B, and a transition relation specified by a transducer T, does there exists a path from I to B through the transition relation $R(T)$? Consider the token-passing protocol again and consider the safety property stating that at most one process owns the token at any time during any run of the system. The set B of bad configurations (those violating the safety property) is given by the regular expression $(t+n)^*t(t+n)^*t(t+n)^*$. Recall that the set of initial configurations I was given by tn^*. The problem amounts to checking whether $(I \circ R^*(T)) \cap B = \emptyset$. The problem can be solved by computing the set $Inv = I \circ R^*(T)$ and checking whether it intersects B. Since $R^*(T) = \{(a,a) \mid a \in \Sigma\} \cup R(T^+)$, to solve the above problem it is sufficient to compute T^+.

Given a transducer T, the transitive closure $R^+(T)$ is not computable in general; and in fact it may not even be finite-state representable. Our goal is to design techniques that try to compute a (finite-state) transducer T^+ that accepts the relation $R^+(T)$ in case such a T^+ exists. We will show one such technique based on *acceleration*.

As a running illustration, we will consider the problem of computing T^+ for the transducer in Fig. 4. A first attempt is to compute T^n that characterizes the

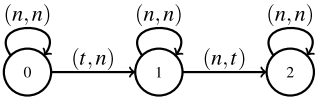


Fig. 7 The transitive closure of T

composition of $R(T)$ with itself n times for $n = 1, 2, 3, \ldots$, i.e., $R(T^n) = R^n(T)$. For example, T^3 characterizes the transition relation where the token gets passed three positions to the right (Fig. 6). A transducer T^+ for $R^+(T)$ is one where the token gets passed an arbitrary number of times, given in Fig. 7.

Obviously, the transducer T^+ cannot be constructed naively by simply computing the approximations T^n for $n = 1, 2, 3, \ldots$, since such an iteration will not converge in a finite number of steps.

Instead, we will propose an alternative solution where we *accelerate* the generation of the different approximations. We will introduce the acceleration technique in two steps. First, we derive an infinite-state transducer T^{col} that accepts $R^+(T)$, which we call the *column transducer*. We will not construct the column transducer explicitly. Instead, in the second step, we will define an equivalence relation $\simeq$ on the set of states of T^{col}, and introduce a procedure that performs *quotienting*, i.e., collapses states that are equivalent with respect to $\simeq$. The quotienting preserves the relation characterized by T^{col} from the collapsed states (see [5] for the details). If the procedure terminates in a finite number of steps, we have achieved our goal of producing a finite-state transducer that accepts the transitive closure of the original transducer.

Column Transducer

Intuitively, T^{col} recognizes the transitive closure $R^+(T)$, i.e., it accepts each pair of words related by $R^i(T)$ for some $i \geq 1$. Recall that a pair of words (x, x') belongs to $R^i(T)$ if x can be rewritten to x' using i successive runs of T. Each such sequence of i runs of T is simulated by a single run of T^{col}. Figure 8 shows a single run of T^{col} that simulates the effect of the sequence of runs $\rho_1, \rho_2, \rho_3, \rho_4$ (in that order) in Fig. 5. The states of T^{col}, called *columns*, are sequences of states in Q (recall that Q is the set of states in the transducer T). The run in Fig. 8 uses six columns, namely $x_0, \ldots, x_5$, each of which is of height 4. In general, T^{col} uses columns of height i

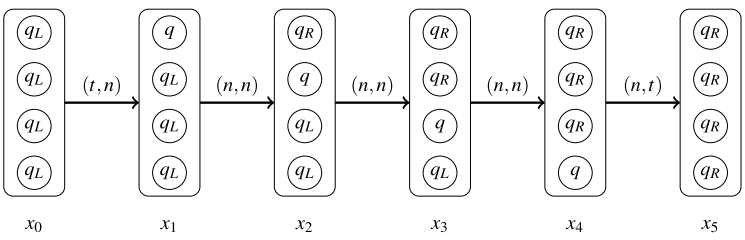


Fig. 8 A run of the column transducer that simulates the four runs in Fig. 5

to create a (single) run that accepts a pair of words in $R^i(T)$. Such a run will be of length k where k is the length of the words in the accepted pair. For instance, the run in Fig. 8 is of length 5. The transitions between the columns of T^{col} are labeled by pairs of symbols that reflect the total effect of the corresponding transitions in the underlying runs of T. For instance, the first transition (between the columns x_0 and x_1) is labeled by (t, n) corresponding to the labels of the first transitions in the runs $\rho_1, \rho_2, \rho_3, \rho_4$. These labels are (t, n) followed by (n, n) (three times) with the total effect of rewriting t to n. Also, the second transition (between the columns x_1 and x_2) is labeled by (n, n) corresponding to the labels of the second transitions in the runs $\rho_1, \rho_2, \rho_3, \rho_4$. These labels are (n, t) follows by (t, n) followed by (n, n) (twice) with the total effect of rewriting n to n. Formally, starting from T, we construct the (infinite-state) transducer $T^{col} = (Q^{\mathtt{col}}, q_{init}^{\mathtt{col}}, \Delta^{\mathtt{col}}, F^{\mathtt{col}})$ where

- $Q^{\mathtt{col}} = Q^+$ is the set of non-empty sequences of states of T.
- $q_{init}^{\mathtt{col}} = q_{init}^+ \subseteq Q^{\mathtt{col}}$ is the set of non-empty sequences of the initial state of T. Notice that the set of initial states of the column transducer is infinite.
- $\Delta^{\mathtt{col}} \subseteq Q^{\mathtt{col}} \times (\Sigma \times \Sigma) \times Q^{\mathtt{col}}$ is defined as follows: for any columns $x_1 = q_1 q_2 \cdots q_m$ and $x_2 = r_1 r_2 \cdots r_m$, and pair (a, a'), we have $(x_1, (a, a'), x_2) \in \Delta^{\mathtt{col}}$ if there exist $a_0, a_1, \ldots, a_m$ with $a = a_0$ and $a' = a_m$ such that $q_i \xrightarrow{(a_{i-1}, a_i)}_T r_i$ for $1 \leq i \leq m$.
- $F^{\mathtt{col}}$ is the set F^+ of non-empty sequences in accepting states of T.

It is easy to see that T^{col} accepts exactly the relation $R^+(T)$: runs of transitions from q_{init}^i (i.e., columns consisting of i copies of q_{init}) to columns in F^i (i.e., columns consisting of i states in F) accept pairs of words that belong to the relation $R^i(T)$.

Quotienting

The problem is that the column transducer has infinitely many states, and therefore it cannot be built explicitly. Instead, we perform *quotienting* based on an equivalence relation $\simeq$ that we define on the set $Q^{\mathtt{col}}$ of columns of T^{col}. We define $\simeq$ in several steps as follows. A state $q \in Q$ is *left-copying* if whenever there exists a run $q_{init} \xrightarrow{(a_0, a_0')}_T q_1 \xrightarrow{(a_1, a_1')}_T \cdots \xrightarrow{(a_{n-1}, a_{n-1}')}_T q_n$ with $q_n = q$, then $a_i = a_i'$ for all $i \in \{0, 1, \ldots, n-1\}$. A *right-copying* state is defined in a similar manner. In other words, prefixes of left-copying states only copy input symbols to output symbols, and similarly for suffixes of right-copying states. In Fig. 4(b), the states q_L and q_R are left- and right-copying respectively. We use Q^{copy} to denote the set of states that are either left- or right-copying. Two columns are *equivalent* if they can be made equal by ignoring repetitions of identical neighbors which are either left- or right-copying. For instance the columns $q_L q_L x q_R$ and $q_L x q_R q_R$ are equivalent. Formally, each equivalence class of $\simeq$ is a set denoted by a regular expression of the form $e_1 e_2 \cdots e_n$ where each e_i is one of the following:

1. q_L^+, for some left-copying state q_L,
2. q_R^+, for some right-copying state q_R,
3. q, for some state q which is neither left-copying nor right-copying.

Furthermore, we require that two consecutive e_i can be identical only if they are neither left-copying nor right-copying. For a column x, let $[x]_{\simeq}$ denote the equivalence class for x. We will use X, Y, etc. to denote equivalence classes of columns.

Having defined the equivalence relation $\simeq$ on $Q^{\mathtt{col}}$, we define a *quotient transducer* $T^\bullet = (Q^\bullet, q^\bullet_{init}, \Delta^\bullet, F^\bullet)$ where

- $Q^\bullet \subseteq Q^{\mathtt{col}}/\simeq$ is a set of equivalence classes of columns.
- $q^\bullet_{init} = q^+_{init}$ is the initial equivalence class (assuming that the initial state is left-copying, this will be one equivalence class of $\simeq$).
- $\Delta^\bullet \subseteq Q^\bullet \times (\Sigma \times \Sigma) \times Q^\bullet$ is defined in the natural way as follows. For any columns x, x' and symbols a, a', if $(x, (a, a'), x') \in \Delta$ then $([x]_{\simeq}, (a, a'), [x']_{\simeq}) \in \Delta^\bullet$.
- $F^\bullet = F^{\mathtt{col}}/\simeq$ is the partitioning of $F^{\mathtt{col}}$ with respect to $\simeq$ (if the final states are right-copying then $F^{\mathtt{col}}$ is a union of equivalence classes).

We will try to build a quotient transducer that accepts the same relation as T^{col} (i.e., $R^+(T)$). Since the transitive closure of $R(T)$ does not need to be recognizable by a finite-state transducer, a finite-state quotient transducer $T^\bullet$ does not necessarily exist. We introduce a procedure that aims at building $T^\bullet$ (Procedure 2) hoping that the procedure terminates in the cases where $T^\bullet$ is finite. The procedure uses two definitions. First, for a state $q \in Q$, we define $q^\oplus := q^+$ if $q \in Q^{copy}$, and define $q^\oplus := q$ otherwise. Second, we define the operator $\star$ as the natural concatenation operator on equivalence classes: $[x]_{\simeq} \star [y]_{\simeq} = [x \cdot y]_{\simeq}$, where $\cdot$ denotes concatenation of columns. It is easy to check that this operation is well-defined. More precisely, let the equivalence classes be represented by their defining regular expressions. Then $(e_1 \cdots e_n) \star (f_1 \cdots f_m)$ corresponds to $e_1 \cdots e_n \cdot f_1 \cdots f_m$, except when e_n and f_1 are both equal to q^+ for some left- or right-copying state q, in which case the expression becomes $e_1 \cdots e_n \cdot f_2 \cdots f_m$. For equivalence classes X and Y, we write $X \xrightarrow{(a,b)}_\bullet Y$ to denote that either (i) $x \xrightarrow{(a,a')}_T y$, $X = x^\oplus$, and $Y = y^\oplus$; or (ii) there are X_1, X_2, Y_1, Y_2, and b such that $X = X_1 \star X_2$, $Y = Y_1 \star Y_2$, $X_1 \xrightarrow{(a,b)}_\bullet Y_1$, and $X_2 \xrightarrow{(b,a')}_\bullet Y_2$.

Our proposed procedure (Procedure 2) incrementally adds new equivalence classes to $Q^\bullet$, and new transitions to $\Delta^\bullet$, and hence the accepted relation will be successively larger subsets of the relation $R^+(T)$. During the algorithm, we maintain a set W of equivalence classes that we have detected (that are reachable from q^+_{init}) but whose successors have not yet been computed. The set W is initialized to contain q^+_{init} (line 1). The algorithm starts iterating until we reach a point where the set W is empty (line 2). During each iteration, we pick and remove an equivalence class X from W. If X has not yet been analyzed (its successors have not yet been computed), then we add all its successors to the set W (line 7), and add all its outgoing transitions to the set $\Delta^\bullet$ (line 8). Furthermore, if it is a final state we will add it to the set $F^\bullet$.

Procedure 2 RMC with Acceleration

Input: Transducer $T = (Q, q_{init}, t, F)$

Output: Transducer $T^\bullet = (Q^\bullet, q^\bullet_{init}, \Delta^\bullet, F)$ such that $R(T^\bullet) = R^+(T)$

1: $q^\bullet_{init} \leftarrow q^+_{init}$; $Q^\bullet \leftarrow \emptyset$; $\Delta^\bullet \leftarrow \emptyset$; $F^\bullet \leftarrow \emptyset$; $W \leftarrow \{q^+_{init}\}$
2: **while** $W \neq \emptyset$ **do**
3: pick and remove some $X \in W$
4: **if** $X \notin Q^\bullet$ **then**
5: $Q^\bullet \leftarrow Q^\bullet \cup \{X\}$
6: **for all** $a, a', Y : X \xrightarrow{(a,a')}_\bullet Y$ **do**
7: $W \leftarrow W \cup \{Y\}$
8: $\Delta^\bullet \leftarrow \Delta^\bullet \cup \{(X, (a, a'), Y)\}$
9: **if** $Y \in F^+ / \simeq$ **then**
10: $F^\bullet \leftarrow F^\bullet \cup \{Y\}$
11: **end if**
12: **end for**
13: **end if**
14: **end while**

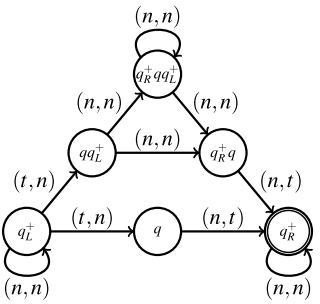


Fig. 9 The transitive closure of T as computed by the algorithm

Example

Figure 9 shows the result of applying our algorithm to the transducer of Fig. 4(b). Below, we describe the main steps.

- First, add q^+_L to W.
- Select q^+_L from W.
  - Since $q^+_L \xrightarrow{(t,n)}_\bullet q$, add q to W, and add $(q^+_L, (t, n), q)$ to $\Delta^\bullet$.
  - $q^+_L \xrightarrow{(t,n)}_\bullet q$ and $q^+_L \xrightarrow{(n,n)}_\bullet q^+_L$ gives $q^+_L \xrightarrow{(t,n)}_\bullet qq^+_L$. Add qq^+_L to W, and add $(q^+_L, (t, n), qq^+_L)$ to $\Delta^\bullet$.
- Select q from W. Since $q \xrightarrow{(n,t)}_\bullet q^+_R$, add q^+_R to W, and add $(q, (n, t), q^+_R)$ to $\Delta^\bullet$. Since $q^+_R \in F / \simeq$ add q^+_R to $F^\bullet$.

- Select q_R^+ from W. Since $q_R^+ \xrightarrow{(n,n)}_\bullet q_R^+$, add q_R^+ to W (since q_R^+ has already been added to $Q^\bullet$, this copy of q_R^+ will be discarded when later selected from W) and add $(q_R^+, (n,t), q_R^+)$ to $\Delta^\bullet$.
- Select qq_L^+ from W:
  - $q \xrightarrow{(n,t)}_\bullet q_R^+$ and $q_L^+ \xrightarrow{(t,n)}_\bullet q$ gives $qq_L^+ \xrightarrow{(n,n)}_\bullet q_R^+q$. Add q_R^+q to W, and add $(qq_L^+, (n,n), q_R^+q)$ to $\Delta^\bullet$.
  - $q \xrightarrow{(n,t)}_\bullet q_R^+$ and $q_L^+ \xrightarrow{(t,n)}_\bullet qq_L^+$ gives $qq_L^+ \xrightarrow{(n,n)}_\bullet q_R^+qq_L^+$. Add $q_R^+qq_L^+$ to W, and add $(qq_L^+, (n,n), q_R^+qq_L^+)$ to $\Delta^\bullet$.
- Select q_R^+q from W. $q_R^+ \xrightarrow{(n,n)}_\bullet q_R^+$ and $q \xrightarrow{(n,t)}_\bullet q_R^+$ gives $q_R^+q \xrightarrow{(n,t)}_\bullet q_R^+$. Add q_R^+ to W (this copy of q_R^+ will be discarded when later selected from W), and add $(q_R^+q, (n,t), q_R^+)$ to $\Delta^\bullet$.
- Select $q_R^+qq_L^+$ from W:
  - $q_R^+q \xrightarrow{(n,t)}_\bullet q_R^+$ and $q_L^+ \xrightarrow{(t,n)}_\bullet qq_L^+$ gives $q_R^+qq_L^+ \xrightarrow{(n,n)}_\bullet q_R^+qq_L^+$. Add $q_R^+qq_L^+$ to W (this copy of $q_R^+qq_L^+$ will be discarded when later selected from W), and add $(q_R^+qq_L^+, (n,n), q_R^+qq_L^+)$ to $\Delta^\bullet$.
  - $q_R^+ \xrightarrow{(n,n)}_\bullet q_R^+$ and $qq_L^+ \xrightarrow{(n,n)}_\bullet q_R^+q$ gives $q_R^+qq_L^+ \xrightarrow{(n,n)}_\bullet q_R^+q$. Add q_R^+q to W (this copy of q_R^+q will be discarded when later selected from W), and add $(q_R^+qq_L^+, (n,n), q_R^+q)$ to $\Delta^\bullet$.

Figure 9 shows the final automaton. Its language is $(n,n)^*(t,n)(n,n)^*(n,t)(n,n)^*$ which is exactly the transitive closure of the language of the transducer in Fig. 4. Computing intersection is easy, so now the safety property is checked by checking whether $(I \circ R^*(T)) \cap B = \emptyset$. Notice that this equality holds in the case of our running example. For the full details of the method, we refer the reader to [5].

21.4 Monotonic Abstraction

Regular model checking is an elegant uniform framework in which one can implement a wide class of techniques. However, it is often necessary to complement it with other techniques in order to make it useful in practice. A limiting factor in the application of transducer-based techniques is the difficulty of computing the transitive closure. As we noticed in the previous section, computing the transitive closure is necessary to solve verification problems such as checking safety properties. However, we saw that such a computation is not guaranteed to terminate in general. Also, the transitive closure may be onerous to compute since it may rely on expensive automata-theoretic constructions. One way to increase the efficiency is to use over-approximations of system behavior. The idea is that we derive a new system which we call the *abstract system* such that the behavior of the abstract system over-approximates the behavior of the original system (the latter is called the *concrete system*). The advantage of such an approach is that it may be possible to

analyze the abstract system more efficiently compared to the concrete system. The price we pay is that we lose some precision. More precisely, the abstract system may contain *false positives*, i.e., violations of the given safety property that do not actually exist in the concrete system. Notice that the method is still sound in the sense that if the abstract system satisfies a safety property, then we can conclude that the concrete system also satisfies the property. In this section, we give an example of an over-approximation which produces an abstract model that is a well quasi-ordered transition system. This allows us to apply the methodology of [2] to the abstract system in the same way as we did in Sect. 21.2.

To illustrate the idea, we consider a new class of parameterized systems where the processes are organized as a linear array. The class includes a new feature, namely that transitions of a process may be constrained by *global conditions*, i.e., the process may have to check the states of all the other processes before performing the transition. Although global conditions are often difficult to analyze, they are an important feature in the behavior of several classes of parameterized systems (such as mutual exclusion protocols). To describe the transitions of the system we will often talk about the *active* process, i.e., the process that is about to perform the next transition, and talk about the *left/right context* of the active process, i.e., all the processes to the left/right of the active process in the configuration. A global transition is either *universally* or *existentially* quantified. An example of a universal condition is that *all* processes in the left context of the active process should be in certain states. In an existential transition we require that *some* (rather than *all*) processes should be in certain states. A parameterized system with universal conditions is able to simulate counter machines (Minsky machines) and therefore all non-trivial verification problems for them are undecidable. Again, we will leave the details of the undecidability proof as an exercise for the reader. In fact, this class of systems can also be modeled and analyzed using transducers as we did in Sect. 21.3. However, we concentrate here on a verification method based on using over-approximations.

21.4.1 Example

We introduce our method through a protocol that implements mutual exclusion among an arbitrary number of processes. Each process (depicted in Fig. 10) has four states, namely the idle (`I`), requesting (`R`), waiting (`W`), and critical (`C`) states.

Initially, all the processes are idle (in state `I`). When a process becomes interested in accessing the critical section (state `C`), it declares its interest by moving to the requesting state `R`. This is described by the global universal transition rule t_1 in which the move is allowed only if all other processes are in their idle or requesting states. The universal quantifier labeling t_1 encodes the condition that all other processes (whether in the left or the right context of the active process—hence the index LR of the quantifier) should be `I` or `R`. In the requesting state, the process may move to the waiting state `W` through the local transition t_2 (the transition is *local* in the sense that the process does not need to check the states of the other processes). Notice

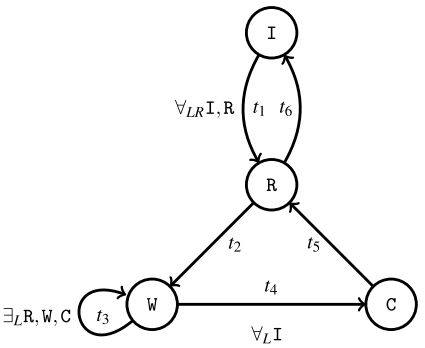


Fig. 10 One process in the mutual exclusion protocol with linear topology

that any number of processes may cross from the idle state to the requesting state. However, once the first process has crossed to the waiting state, it "closes the door" on the processes which are still in their idle states. These processes will no longer be able to leave their idle states until the door is opened again (when no process is in W or C). From the set of processes that have declared interest in accessing the critical section (those that have left their idle states and are now in the requesting or waiting states) the leftmost process has the highest priority. This is encoded by the universal transition t_4 where a process may move from its waiting state to its critical section subject to the universal condition that all processes in its left context are idle. If the process finds out that there are other processes that are requesting, in their waiting states, or in their critical sections, then it loops back to the waiting state through the existential transition t_3. Once the process leaves the critical section, it will return to the requesting state through the local transition t_5. In the requesting state, the process chooses either to try to reach the critical section again, or to become idle (through the local transition t_6).

21.4.2 Model

Since the system has a linear topology, its configurations are of the same form as those in Sect. 21.3. More precisely, a configuration is represented as a word over a finite alphabet representing the local states of the processes. In our example, this alphabet is given by the set {I, R, W, C}. For instance the configuration IWCWR represents a configuration in an instance of the system with five processes that are in their idle, waiting, critical, waiting, and requesting states in that order. The definition of the transition relation $\xrightarrow{t}$ depends on the type of transition t (whether it is local, existential, or universal). We will consider three transition rules from Fig. 10 to illustrate the idea. The local rule t_2 induces transitions of the form WIRCR $\xrightarrow{t_2}$ WIWCR. Here the active process changes its state from requesting to waiting. The existential rule t_3 induces transitions of the form RIWCR $\xrightarrow{t_3}$ RIWCR. The waiting process can perform the transition since there is a requesting process in its left context. However, the same transition is not enabled from the configuration IIWCR, since there

are no critical, waiting, or requesting processes in the left context of the process trying to perform the transition. The universal rule t_4 induces transitions of the form $\texttt{IIWWR} \xrightarrow{t_4} \texttt{IICWR}$. The active process (in the waiting state) can perform the transition since all processes in its left context are idle. On the other hand, from the configuration CIWWR, neither of the waiting processes can perform the transition since, for each one of them, there is at least one process in its left context which is not idle.

An *initial configuration* is one in which all processes are in their idle state. Examples of initial configurations are II and IIIII, corresponding to instances of the system with two and five processes respectively. As mentioned above, the protocol is intended to guarantee mutual exclusion. The set of bad configurations consists of all configurations that contain at least two processes in their critical sections. Examples of bad configurations are CRC and ICRCWC. As before, showing the safety property amounts to proving that the protocol, starting from an initial configuration, will never reach a bad configuration.

21.4.3 Over-Approximation

Given a parameterized system such as the one described above, we generate an abstract system that (i) is an over-approximation of the concrete system, and (ii) induces a well quasi-ordered transition system in the sense of Sect. 21.2. We define an ordering on configurations where $c_1 \preceq c_2$ if c_1 is a (not necessarily contiguous) subword of c_2. For instance, $\texttt{WC} \preceq \texttt{RWICW}$. The relation $\preceq$ is a wqo by Higman's lemma [40]. The aim of our approximation is to produce a transition relation that is monotonic with respect to $\preceq$ (thus satisfying the first of the five needed properties in Sect. 21.2.4). We will first motivate why local and existential transitions are actually monotonic.

Consider the local rule t_2 and the induced transition $c_1 = \texttt{IRC} \xrightarrow{t_2} \texttt{IWC} = c_2$ in which a process changes state from requesting to waiting. Consider the configuration $c_3 = \texttt{IWIRCR}$ that is larger than c_1. Clearly, c_3 can perform the local transition $c_3 = \texttt{IWIRCR} \xrightarrow{t_2} \texttt{IWIWCR} = c_4$ leading to $c_4 \succeq c_2$. Local transitions are monotonic, since the active process in the smaller configuration (the requesting process in c_1) also exists in the larger configuration (i.e., c_3). A local transition does not check or change the states of the passive processes; hence the larger configuration c_3 is also able to perform the transition, while maintaining the ordering $\preceq$.

Consider the rule t_3 and the induced transition $c_1 = \texttt{RIWCR} \xrightarrow{t_3} \texttt{RIWCR} = c_2$. Observe that the configuration c_1 can be divided into three parts: the active process in state W, the left context RI, and the right context CR. Furthermore, the left context contains a witness (the process in state R) which enables the transition. Consider the configuration $c_3 = \texttt{IRIWCRC}$ that is larger than c_1. Again, the configuration c_3 can be divided into three parts: the active process in state W, the left context IRI, and the right context CRC. Notice that the left and right contexts of the active process

in c_3 are larger than their counterparts in c_1. In particular, the left context in c_3 will also contain the witness. This means that c_3 can perform the same transition $c_3 = \texttt{IRIWCRC} \xrightarrow{t_3} \texttt{IRIWCRC} = c_4$ leading to $c_4 \succeq c_2$.

Next, we motivate why universal transitions are not monotonic. Consider the universal rule t_4 and the induced transition $c_1 = \texttt{IIWWR} \xrightarrow{t_4} \texttt{IICWR} = c_2$. The transition is enabled since all processes in the left context of the active process satisfy the condition of the transition (they are idle). Consider the configuration $c_3 = \texttt{IRIRWWR}$. Although $c_1 \preceq c_3$, the universal transition t_4 is not enabled from c_3 since the left context of the active process contains processes that violate the condition of the transition. This means that universal transitions are not monotonic.

Since local and existential transitions are monotonic, they need not be approximated. Therefore, we only provide an over-approximation for universal transitions. We define a new transition relation $\rightsquigarrow$ such that $\xrightarrow{t} \subseteq \overset{t}{\rightsquigarrow}$ for each transition t. If t is local or existential, we define $\overset{t}{\rightsquigarrow} := \xrightarrow{t}$, i.e., $\rightsquigarrow$ agrees with $\longrightarrow$ on local and existential transitions. In order to deal with non-monotonicity of universal transitions, we will define $\overset{t}{\rightsquigarrow}$, for a universal transition t, such that $\xrightarrow{t} \subseteq \overset{t}{\rightsquigarrow}$. Intuitively, we perform $\overset{t}{\rightsquigarrow}$ by first deleting all the processes violating the condition of the universal rule, and then performing $\xrightarrow{t}$. This means for instance that we have a transition of the form $\texttt{IRIRWWR} \overset{t_4}{\rightsquigarrow} \texttt{IICWR}$. The reason is that we can first delete the two processes in the requesting states, thus obtaining IIWWR, and then perform $\xrightarrow{t_4}$ from IIWWR, which gives IICWR. The approximate transition relation $\rightsquigarrow$ is also monotonic with respect to universal transitions.

21.4.4 Backward Reachability Algorithm

We show that the transition system induced by the set of configurations and the abstract transition relation $\rightsquigarrow$ satisfies the five conditions needed for the applicability of the backward reachability algorithm (as described in Sect. 21.2.4). First, the ordering $\preceq$ is a wqo by Higman's lemma (see above), and it can be checked (this amounts to checking whether a finite word is a subword of another). The transition relation $\rightsquigarrow$ is monotonic with respect to $\preceq$ as described above. For a finite set *Bad* of configurations and an initial set C_{init} of configurations (described, e.g., as a regular expression), checking the emptiness of $\widehat{Bad} \cap C_{init}$ amounts to checking the emptiness of the intersection of two regular expressions. The only property that remains to be shown is computability of the predecessor set. For a configuration c and a transition rule t, we define $Pre(t)(c)$ to be the set $\{c_1, \ldots, c_n\}$ which is the generator of the set of configurations from which we can reach $\widehat{c}$ through one application of $\overset{t}{\rightsquigarrow}$ (in a similar manner to Sect. 21.2.4). The formal definition of how to compute $Pre(t)(c)$ can be found in [3]. Here, we give an informal explanation, and will consider different transition rules in Fig. 10 to illustrate how to compute *Pre*.

For a local transition, we simply run this transition backwards on the active process. For instance, for the local rule t_5 in Fig. 10, we have $Pre(t_5)(\texttt{IRW}) = \{\texttt{ICW}\}$.

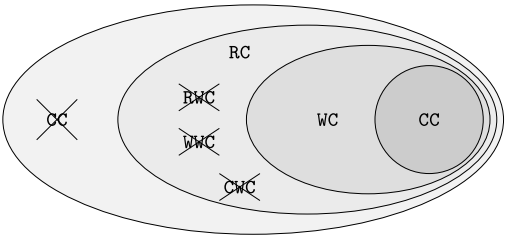


Fig. 11 Running the backward reachability algorithm on the example protocol

In other words, the predecessor set is characterized by one configuration, namely ICW. Strictly speaking, the set also contains a number of other configurations such as IRCW. However such configurations are subsumed by the original configuration IRW, and therefore for the sake of simplicity we will not include them in the set.

For existential transitions, there are two cases depending on whether a witness exists or not in the configuration. Consider the existential rule t_3 in Fig. 10. We have $Pre(t_3)(\texttt{RWC}) = \{\texttt{RWC}\}$. In this case, there is a witness (a process in state R) in the left context of the active process. On the other hand, we have $Pre(t_3)(\texttt{IWC}) = \{\texttt{RIWC}, \texttt{IRWC}, \texttt{WIWC}, \texttt{IWWC}, \texttt{CIWC}, \texttt{ICWC}\}$. In this case there is no witness in the left context of the active process. Therefore, we add a witness explicitly in each possible state (R, W, or C), at each possible place in the left context of the active process. Notice that the size of the new configurations (four processes) is larger than the size of the original configuration (three processes). This means that the size of the configurations generated by the backward algorithm may increase, and hence there is *a priori* no bound on the size of these configurations.

For universal conditions, we check whether there is any process in the configuration violating the condition. Consider the universal rule t_4 in Fig. 10. Then $Pre(t_4)(\texttt{IRICW}) = \emptyset$ since there is a requesting process in the left context of the potential active process (which is in the critical section). On the other hand, $Pre(t_4)(\texttt{IICW}) = \{\texttt{IIWW}\}$ since all processes in the left context of the active process are in their idle states.

Since the five needed properties are satisfied, we can now run the backward reachability algorithm of Sect. 21.2.4 over our class of parameterized systems. We show how the algorithm runs on our example (Fig. 11). We start with the generator of the set of bad configurations, namely {CC}. The only transition that is enabled backwards from a critical state is the one induced by the rule t_4. Of the two processes in CC only the left one can perform t_4 backwards (the right process cannot perform t_4 backwards since its left context contains a process not satisfying the condition of the quantifier): $Pre(t_4)(\texttt{CC}) = \{\texttt{WC}\}$. From WC, two rules are enabled backwards (both from the process in state W): the local rule t_2: $Pre(t_2)(\texttt{WC}) = \{\texttt{RC}\}$; and the existential rule t_3: $Pre(t_3)(\texttt{WC}) = \{\texttt{RWC}, \texttt{WWC}, \texttt{CWC}\}$. All three configurations in $Pre(t_3)(\texttt{WC})$ are subsumed by WC. One rule is enabled backwards from RC, namely the local rule t_5 from the requesting process: $Pre(t_5)(\texttt{RC}) = \{\texttt{CC}\}$. Notice that the universal transition t_1 is not enabled from the requesting process, since there exists another process (the process in state C) in the configuration that violates the condition of the quantifier. At this point, the algorithm terminates, since it is not possible to provide any new configurations that are not subsumed by the existing ones.

Since there is no initial configuration (with only idle processes) in $\widehat{\mathsf{CC}} \cup \widehat{\mathsf{WC}} \cup \widehat{\mathsf{RC}}$, the set of bad configurations is not reachable from the set of initial configurations in the abstract system. Since $\longrightarrow \subseteq \rightsquigarrow$, the set of bad configurations is not reachable from the set of initial configurations in the concrete system either. For the full details of the method, we refer to [3].

21.5 Compositional Reasoning for Parameterized Verification

In this section we present an abstraction-and-compositional-reasoning-based method called the *CMP Method* (abbreviation for *CoMPositional*) that has been successful in parametrically verifying safety properties for industrial-strength distributed protocols. Unlike the methods presented in the previous sections, this method does not aim at full automation but rather at scalability. The key idea behind this technique is to reduce replication in a parameterized system to a minimum using lightweight but very coarse abstractions. The resulting abstract model is refined progressively using manually supplied *non-interference lemmas* (or candidate invariants) until it either passes the check or a real counterexample is found for the property under verification. The beauty of the method is that it works with abstract models of the smallest possible size and when the proof is done the invariants added by the user don't have to be inductive. Thus, it is scalable and keeps the burden on the user to a minimum.

21.5.1 Parameterized Protocols

We will formalize our parameterized systems differently in this section so that they correspond more closely to how systems are actually implemented in practice. A parameterized protocol $P(N)$ is a protocol with N agents with unique identifiers (ids) in $\mathbb{N}_N = [1..N]$. A parameterized protocol is called *symmetric* if the behaviors of the agents are similar and independent of their id value. More formally, let R_i denote the reachable states of $P(N)$ in i steps and $\phi(1..M)$ be a safety property over the agents $1..M$, $M \leq N$. $P(N)$ is symmetric if for all $i \geq 0$ and $M \leq N$

$$\forall s \in R_i.s \models \phi(1, .., M) \quad \Rightarrow \quad \forall s \in R_i.s \models \forall i_1, .., i_M.\phi(i_1, .., i_M).$$

That is, if an indexed formula holds for one combination of agents then it holds for every combination. Following the standard notation we write $P(N) \models \phi(1, .., M)$ if all reachable states of $P(N)$ satisfy $\phi(1, .., M)$ (that is, $\forall i \geq 0. \forall s \in R_i.s \models \phi(1, .., M)$). From the above definition of symmetry we have the following:

$$P(N) \models \phi(1, .., M) \quad \Rightarrow \quad P(N) \models \forall i_1, .., i_M.\phi(i_1, .., i_M).$$

To keep the presentation clean we will use a simple model sufficient to express cache coherence protocols, which were the motivating examples for this work, *but*

the correctness of the CMP method depends only on the protocol being symmetric and described with guarded commands.

Index Variables

Fix a set $\mathscr{I}$ of index variables for quantifying over process ids. When we write $T(i, j)$, we mean that i and j are the only index variables appearing free in $\mathscr{I}$.

States

The state of a protocol consists of global and local variables that can hold either Boolean values or process ids. A global variable (such as a directory entry) is a scalar variable, and a local variable (such as a cache entry) is an array variable indexed by process id. Formally, the state is determined by four sets of variables, W, X, Y, Z, where variables in W are of Boolean type denoted by $\mathbb{B}$, those in X are of type $\mathbb{N}_N \to \mathbb{B}$, those in Y are of type $\mathbb{N}_N$ and those in Z are of type $\mathbb{N}_N \to \mathbb{N}_N$. Note that the types of the variables are determined by the parameter N.

Expressions

An expression is made up of combination of the four *basic expressions*

$$w, \quad y = i, \quad x[i], \quad \text{and} \quad z[i] = j$$

where $i, j \in \mathscr{I}$ are index variables and $w \in W, x \in X, y \in Y, z \in Z$ in the usual fashion by taking their Boolean combinations and possibly quantifying out some free index variables, that is, first-order formulas over the basic expressions. Observe that we have not permitted expressions such as $i \leq j$ or $z[i] \leq z[j]$ that introduce asymmetry by making the index value significant. To keep the presentation simple we avoid such expressions, but the method presented here can be generalized to protocols like Bakery [49] or Szymanski's mutual exclusion algorithm [65], which do have them.

Assignments

An assignment is one of the four *basic assignments*

$$w := b \quad y := i \quad x[i] := b \quad z[i] := j$$

where $i, j \in \mathscr{I}$, b is either *true* or *false*, and $w \in W, x \in X, y \in Y, z \in Z$, together with the *quantified assignment*

$$\forall k.\phi(k) \Rightarrow \left[v[k] := e\right]$$

where $v[i] := e$ is one of the basic assignments and $\phi(k)$ is an expression as described above. The latter is just a set of conditional assignments: consider all possible assignments of values to index i, and for each such assignment, perform $v[i] := e$ if $\phi(i)$ is true.

Rules

A rule

$$rl(i, j) : \rho(i, j) \rightarrow a(i, j)$$

is a *guarded command* where rl is the *rule name*, ρ is an expression called the *guard*, and a is a list of assignments called the *action*. Process indices i and j are the only free index variables in ρ and a. We further assume that an action never assigns a variable more than once. This rules out asymmetric assignments like $\forall k.true \Rightarrow y := k$ where the value received by y depends on the order in which the $\forall$ quantifier is evaluated.

A rule set

$$rl : \forall i, j.\rho(i, j) \rightarrow a(i, j)$$

is just a compact notation for representing a set of identical rules differing only in the indices. Defining transitions using rule sets ensures that all agents have the same set of rules available.

Protocols

A *protocol* is a state transition system (S, Θ, T), where S and Θ are the sets of states and initial states and $T \subseteq S \times S$ is the transition relation given by a set of rule sets. There is a transition from s to s' if T contains a rule $rl(p, q) : \rho(p, q) \rightarrow a(p, q)$ for $p, q \in \mathscr{I}$ such that s satisfies $\rho(p, q)$ and s' is the result of starting with s and performing the assignments in $a(p, q)$.

From these restrictions on expressions and assignments it is clear that the protocol P must be symmetric.

Safety Properties

A safety property is a first-order formula over the basic expressions defined above in addition to equality expressions $i = j$ over indices in $\mathscr{I}$. We further assume no index variable appears free in the safety property. A typical safety property for cache protocols is the coherence property

$$\forall i, j.i \neq j \Rightarrow \big(exclusive[i] \Rightarrow \neg exclusive[j]\big)$$

Fig. 12 CMP(P, I) for protocol $P(N)$ and property I

```
1:  P# = P; I# = I
2:  while A(P#) ⊭ A(I#(1,2)) do
3:      examine counterexample cex
4:      if cex is a real counterexample then
5:          return cex
6:      else
7:          find L = ∀i, j.L(i, j) ruling out cex
8:          P# = strengthen(P#, L)
9:          I#(i, j) = I#(i, j) ∧ L(i, j)
10:     end if
11: end while
12: return I is invariant
```

where $exclusive[i]$ is a predicate over state variables of i such that $exclusive[i] = true$ means i has exclusive access to the data item. In practice, we usually see two-indexed or three-indexed properties. For the rest of the chapter, we assume we have two-indexed properties only. Note that there are no temporal operators in our notion of safety properties.

21.5.2 The CMP Method

The CMP method consists of two basic steps—*abstraction* and *strengthening*—that are applied iteratively to a protocol. The abstraction procedure used in CMP retains detailed information on a small number of processes, and abstracts away the remaining processes. Since our protocols are symmetric there is no loss of generality in focussing on processes 1 and 2.[2] Given a symmetric protocol P with N processors $[1..N]$ and a safety property $\forall i, j \in [1..N].I(i, j)$, the method is shown in Fig. 12 below. We assume that the abstraction procedure $A()$ is sound, that is, if $A(P^{\#}) \models A(I^{\#}(1, 2))$ then $P^{\#} \models I^{\#}(1, 2)$.

Except for step 7 of the algorithm in Fig. 12, which requires the user to add *lemmas* (or expressions that are potentially invariants), all the other steps of the CMP method are automatic. If the loop terminates normally, we conclude that $I^{\#}$ and consequently I are invariants of P; see Sect. 21.5.2.4 for the justification. The final $I^{\#}(i, j)$ is the conjunction of the initial safety property $I(i, j)$ with all the user-added lemmas $L(i, j)$. If the loop terminates via the exit (line 5), then either I or one of the proposed lemmas L is not an invariant of the protocol, and the user must back up and try again.

Remark 1 Note that the CMP method follows the *abstract-and-refine* paradigm but it differs from predicate abstraction. The latter consists of enlarging the set of predicates used in the abstraction operation so as to obtain finer resolution. The abstract model ends up having a larger state space after each refinement step. But in CMP

[2] For three-indexed properties we would focus on processes 1, 2, and 3.

the abstraction operation is unchanged. Rather, the refinement happens by *strengthening* of the protocol itself—that is, by a syntactic modification of the protocol code. The abstract model has a *smaller* state space after each refinement step.

In the rest of the section we will describe in more detail the abstraction and strengthening operations and prove that the CMP method is sound.

21.5.2.1 German's Protocol

We will use German's protocol, a well-known academic directory-based cache coherence protocol, to illustrate the different steps of the CMP method. In this distributed system, there is a collection of identical processors with ids drawn from $[1..N]$ and a central directory that co-ordinates access to data items. A processor can hold a data item d in $E, S,$ or I, standing for *exclusive* access, *shared* access, and *invalid* or no access, respectively. In the exclusive state, only processor i has access to d, so it can modify it. In the shared state, other processors can access d as well, so none of the sharers can modify it.

Each processor has three channels to the directory: $Chan1$, which carries request messages from processor to directory, $Chan2$, which carries grant messages from directory to processor, and $Chan3$, which carries invalidate messages. The difference between $Chan2$ and $Chan3$ is that the former carries responses from the directory while the latter carries messages that are initiated by the directory.

We simplify this toy example further and assume there is only data item in the system. So we can talk about the state of a processor i instead of the state of data item d in i. If processor i wants to transition from state I to state S, then it sends a $ReqS$ message to the directory on $Chan1$. If no other processor holds the data item then the directory responds with a $GntS$ message on $Chan2$. In case processor j has exclusive access then the directory first invalidates j's access by sending it an Inv message on $Chan3$. Once j invalidates its access and sends back an $InvAck$ message, the directory sends a $GntS$ to i. Transactions for gaining exclusive access are similar except that more than one processor needs to be invalidated if there are multiple sharers.

This is perhaps the simplest possible directory-based cache coherence protocol and it has been studied widely using various types of model checkers including Murphi [41]. The work presented has also been used in conjunction with Murphi to verify the correctness of German's protocol. The Murphi description is straightforward with states of processors modeled using arrays. The channels are also modeled as arrays, i.e., $Chan1$ is modeled as an array $Chan1[1..N]$.

21.5.2.2 Abstraction

A sound abstraction is a procedure that transforms one protocol P into another protocol $\hat{P} = A(P)$, and transforms one property ϕ over states of P into another property $\hat{\phi} = A(\phi)$ over states of $\hat{P}$ such that $\hat{P} \models \hat{\phi} \Rightarrow P \models \phi$. Apart from being sound,

we make the additional requirement that the abstraction focuses on two processes 1 and 2 and abstracts the rest away.[3]

One such well-known abstraction is *data type reduction* [53], which reduces data types with large or unbounded ranges to ones with small, finite ranges. Given a variable v with a large range, say $[1..L]$, it can be syntactically abstracted to a variable $\hat{v}$ with a small range $\{1, 2, o\}$ which retains two values, 1 and 2, and the rest of the values are lumped into an abstract value o.

We denote the syntactic abstraction operation by A_{red}; the abstract protocol $A_{red}(P)$ is constructed via a data type reduction which retains a small number of processors, say processors 1 and 2, and replaces the remaining processors 3, .., N with a single, highly nondeterministic process called the $Other$ process. The abstraction process is a simple, syntactic procedure: any condition in the protocol code involving processors 3, .., N is replaced with *true* or *false* to *over-approximate* it depending on whether it occurs in a positive or negative context. If it occurs in both negative and positive contexts, then it is replaced with a nondeterministic Boolean variable. A nondeterministic variable simply allows the model checker to try out both *true* and *false* values during the verification and is just a convenient mechanism to keep the description of the abstract model from becoming too large. Any assignment to the state variables of $[3..N]$ is deleted. The abstraction $A_{red}(\phi)$ of a safety property ϕ is defined similarly except that any condition involving processors $3..N$ in property ϕ is replaced with *true*, *false*, or a nondeterministic Boolean variable as appropriate to conservatively *under-approximate* it. Data type reduction is purely syntactic which means it is very fast to compute the abstract model.

This abstraction is best explained with an example. Consider the following rule set from the Murphi description [41] of German's protocol. This rule set is just a collection of guarded commands indexed by a process id $i \in$ NODE, where NODE is the parameterized range $[1..N]$:

```
rule set i : NODE; do
  rule "RecvGntS"
    Chan2[i].Cmd = GntS
    ==>
      begin BODY endrule;
endruleset;
```

In our notation this would be

$$RecvGnts : \forall i \in NODE.Chan2[i].Cmd = GntS \rightarrow BODY.$$

The action of the rule, called *BODY*, is left unspecified as we focus only on the guard. We replace this rule set with N independent rules and apply data type reduction to each rule independently. For processors 1 and 2, the rules remain unchanged (ignoring the effect of data type reduction on the body of the rule). That is, the rules

[3] Generalization to more than two processes is simple.

for processors 1, 2 would be given by

$$RecvGnts : \forall i \in [1,2].Chan2[i].Cmd = GntS \rightarrow BODY.$$

But for processors $i > 2$ the abstracted rule becomes

$$AbsRecvGntS : true \rightarrow BODY'.$$

This is because the condition $Chan2[i].Cmd = GntS$ refers to the state variable of an abstracted process and it is conservatively over-approximated to *true*. Thus, after applying data type reduction to the rule set we will end up with two rule sets: the abstract rule shown above and a rule set identical to the original rule set except that the quantifier i ranges over [1..2]. The example gives a flavor of the syntactic nature of data type reduction and it should be clear that the abstract rules obtained from data type reduction are sound abstractions.

Theorem 1 *The abstraction A_{red} is sound for $\phi(1,2)$ for every expression $\phi(i,j)$:*

$$A_{red}(P) \models A_{red}\big(\phi(1,2)\big) \;\Rightarrow\; P \models \phi(1,2).$$

We refer the reader to McMillan [53] and Krstic [46] for a more detailed treatment of data type reduction.

21.5.2.3 Strengthening

A *strengthening* is a procedure that transforms one protocol P into another protocol $P^{\#} = strengthen(P, \psi)$ by replacing each guarded command $\rho \rightarrow a$ of P with the guarded command $\rho \wedge \psi \rightarrow a$ whose guard has been strengthened by ψ. Given a property ϕ, a strengthening is said to be *sound for* ϕ if it satisfies the property

$$P^{\#} \models \phi \Rightarrow P \models \phi.$$

Returning to the abstraction of the *RecvGntS* rule set, the abstract rule is clearly too abstract. The guard *true* does not constrain the $Other$ process in any way. This leads to spurious counterexamples, and to eliminate such counterexamples, CMP depends on user-provided non-interference lemmas. Suppose we have the following lemma that we think might be useful:

$$\forall p, i \in NODE.Chan2[i].Cmd = GntS \Rightarrow \big(i \neq p \Rightarrow Cache[p].State \neq E\big).$$

This lemmas says if process i has an incoming grant share access message (*GntS*) then all other agents $p \neq i$ must be in non-exclusive states. Strengthening the protocol with this lemma—applying it to the *RecvGntS* rule set—and abstracting, we get the following rule set for the concrete processors:

$$RecvGntS : \forall i \in NODE.Chan2[i].Cmd = GntS \wedge$$
$$\forall p \in NODE.i \neq p \Rightarrow \big(Cache[p].State \neq E\big) \rightarrow BODY$$

and the following rule set for the abstracted process

$$AbsRecvGntS : \forall p \in [1,2].\big((Cache[p].State \neq E) \rightarrow BODY'\big).$$

Note that the abstract rule now has a meaningful guard and thus the abstract rule is more refined than previously. Further, no new state is added to the abstract model during the refinement. Only the set of transitions is pruned. This process is continued iteratively by adding more and more lemmas until either a real counterexample is found or all the lemmas and the property of interest are proved.

21.5.2.4 Correctness

Like other assume-guarantee style methods [1, 20, 46, 52, 55], the CMP method seems circular. In particular, the strengthening operation modifies the protocol by assuming the very invariant we want to prove. The following lemma proves that this circularity can be broken by induction over time.

Lemma 1 *Let R_i be the states of P reachable within i steps, and let $P^{\#} = strengthen(P, \psi)$. If*

$$\forall i.(\forall s \in R_i.s \models \phi) \;\Rightarrow\; (\forall s \in R_i.s \models \psi)$$

then $P^{\#} \models \phi$ implies $P \models \phi$.

Proof Denote by $R_i^{\#}$ the set of all states reachable in $P^{\#}$ within i steps. We will prove by induction on i that $\forall s \in R_i.s \in R_i^{\#}$ and consequently $\forall s \in R_i.s \models \phi$.

For $i = 0$, if s is an initial state of P, then it is an initial state of $P^{\#}$ as well. So the base case for induction is true.

Assume that we have proved the inductive hypothesis for $i = k$. That is, $\forall s \in R_k.s \models \phi$ and $\forall s \in R_k.s \in R_k^{\#}$. We will prove that $\forall s \in R_{k+1}.s \models \phi$ and consequently $\forall s \in R_k.s \in R_k^{\#}$.

Consider any state $s' \in R_{k+1}$ reachable from $s \in R_k$ via a rule $\rho \rightarrow a$. For the rule to fire we must have $s \models \rho$. By the inductive hypothesis, $s \in R_k^{\#}$ as well. Moreover, from $\forall s \in R_k.s \models \phi$ and the condition $\forall i.(\forall s \in R_i.s \models \phi) \Rightarrow (\forall s \in R_i.s \models \psi)$ we have $\forall s \in R_k.s \models \psi$. Consequently, we have $s \models \psi$. Putting all the facts together, we have s is reachable in $P^{\#}$ within k steps and the rule $\rho \wedge \psi \rightarrow a$ is enabled at s. Therefore, s' is reachable in $P^{\#}$ within $k+1$ steps. Since $P^{\#} \models \phi$ we immediately have $s' \models \phi$. □

We use the phrase *entailment* to refer to the condition

$$\forall i.(\forall s \in R_i.s \models \phi) \Rightarrow (\forall s \in R_i.s \models \psi).$$

It is because of this notion of entailment that our lemma for compositional reasoning given above differs subtly but significantly from the compositional reasoning

principles considered by McMillan [52], Abadi and Lamport [1], Krstic [46], Bhattacharya et al. [11] and Chen et al. [19]. In our case, in proving a property ϕ not only can we assume ϕ, but also the "meta-consequence" ψ of ϕ. If $\phi \Rightarrow \psi$ then ψ is a logical consequence of ϕ and entailment clearly holds. But entailment can hold even if ψ is not a logical consequence as in the case of symmetric systems. If property $I(1, 2)$ holds for reachable states R_i then by symmetry $I(3, 4)$, $I(4, 5)$,.. must hold for R_i as well. Properties $I(3, 4)$,.. are not logical consequences of $I(1, 2)$ but are what we call meta-consequences of $I(1, 2)$. They follow from some higher-level property of the system. It is clear the set of meta-consequences is much richer than that of logical consequences and it frees up compositional reasoning to incorporate domain knowledge, e.g., symmetry in our case. Note that ψ does not have to be discharged explicitly once the entailment condition is established.

Theorem 2 *Given a symmetric parameterized system $P(N)$, a property $\forall i, j.I$, and a sound abstraction procedure $A()$, if $CMP(P, I)$ terminates with a proof then $P(N) \models \forall i, j.I$*

Proof Since the protocol is symmetric, we have the entailment precondition

$$\forall i.\big(\forall s \in R_i.s \models I^{\#}(1, 2)\big) \Rightarrow \big(\forall s \in R_i.s \models \forall j, k.I^{\#}(j, k)\big)$$

of Lemma 1. Thus,

$$A(P^{\#}) \models A(I^{\#}(1, 2)) \Rightarrow P^{\#} \models I^{\#}(1, 2)$$

by the soundness of abstractioni, and

$$P^{\#} \models I^{\#}(1, 2) \Rightarrow P \models I^{\#}(1, 2)$$

by the soundness of strengthening (Lemma 1), and

$$P \models I^{\#}(1, 2) \Rightarrow P \models \forall j, k.I^{\#}(j, k)$$

by symmetry. □

The significance of this analysis is that it shows that CMP is sound for any abstraction procedure $A()$ and not just data type reduction A_{red}. Prior work [20, 46] proved

$$A_{red}(P^{\#}) \models A_{red}\big(I^{\#}(1, 2)\big) \Rightarrow P \models \forall i, j.I^{\#}(i, j)$$

with a single, complex proof that depended heavily on symmetry and the use of data type reduction as the abstraction procedure. But the soundness of strengthening depends only on entailment (the hypothesis of Lemma 1) which happens to be satisfied by symmetric protocols. Realizing this, we can prove the soundness of strengthening independently of the specific abstraction procedure being used.

Remark 2 Note that, unlike the usual counterexample-guided refinement approaches, CMP requires the abstract model be sound only for $I^{\#}(1,2)$ and not for the full property $\forall i, j.I^{\#}(i,j)$. This means the abstraction can record much less information and thus scale to larger examples.

21.5.3 Evolution of the CMP Method

Compositional reasoning has been recognized as a natural way to decompose proofs of systems made of up multiple components, both for manual and machine-assisted proofs. While there have been several assume-guarantee principles that avoided circularities—that is, assuming the very property to be proved—most practically useful ones necessarily have it. The earliest circular compositional-reasoning principle was proposed by Chandy and Misra [55]. Abadi and Lamport [1] proposed compositional-reasoning principles that could handle fairness assumptions in addition to the usual safety properties for systems with interleaving semantics. They were also the first to use compositional reasoning in conjunction with decision procedures to automate proofs as far as possible.

McMillan [52] gave a compositional-reasoning principle that applies to synchronous systems as well, that is, assumptions can constrain the system behavior in the same cycle not just in the next cycle (as in interleaving semantics). In the same paper and a later one [54], he showed how compositional reasoning and abstraction can be combined to prove safety properties of complex parameterized systems using the SMV model checker as a proof assistant. The work by McMillan [52, 54] and subsequent formalization of it by Chou et al. [20] and Krstic [46] forms the basis for the method presented here and the term *CMP* is broadly used to refer to them as well.

Looking back at the definition of the CMP method in Fig. 12, in proving $I(1,2)$ we are assuming $\forall i, j.I(i,j)$. This differs from other compositional-reasoning principles, which can only assume $I(1,2)$. Further, the CMP loop presented in Fig. 12 allows us to use any sound abstraction unlike the earlier works [20, 46, 52, 54] which were all restricted to using data type reduction. Thus in its current state CMP is a powerful method that allows any sound abstraction to be used within a very flexible compositional-reasoning framework.

21.5.4 Applications

Because data type reduction retains only two processors and abstracts away the rest, CMP can scale to very large protocols that cannot be handled by other methods. In fact, the main bottleneck in the CMP method is not the model-checking time but rather the effort required by the user to come up with non-interference lemmas. Even this burden is significantly lower compared to theorem-proving methods such as [57] because the invariants added by the user don't have to be inductive.

While several methods have been used to verify German's cache coherence protocol, only two or three methods, all based on theorem proving, have been able to verify the coherence property for the Flash cache coherence protocol. The CMP method was one of the first to verify this protocol parametrically [20, 54]. The proof described in [20] took just a few hours (of human user plus machine time) to finish compared to theorem-proving-style efforts in [24, 57] which took a few days.

CMP has been used to verify the coherence property of an industrial-strength cache protocol several orders of magnitude larger than even the Flash protocol [66]. The number of distinct message types in a cache protocol determines the number of different states of each processor and also the number of different interleavings possible during execution. Thus, it is a good indicator of the cache protocol's complexity. German's protocol, a small-sized protocol, has seven different message commands. The flash protocol, which is a medium-sized protocol, has 16. The industrial protocol considered in [66] has 54 different message types and consequently is orders of magnitude more complex than even Flash. The proof required adding 25 non-interference lemmas over a period of a month. Only the CMP method has been able to handle protocols of this size.

Even for this large industrial protocol, the model-checking time itself was only 4 to 5 hours per run, so the main bottleneck was the manual discovery of lemmas [56, 66]. In [67] Talupur and Tuttle describe a new way of deriving powerful invariants from high-level information about protocols. They show how convergence of the CMP loop can be accelerated using auto-generated lemmas. The resulting augmented CMP method places significantly less burden on the user than the plain CMP method. The LCP protocol considered in [56] is comparable in size to the cache protocol considered in [66] but its proof using the augmented CMP method required only five manually discovered lemmas. This extension is crucially dependent on the ability to use richer abstractions than just data type reduction. Thus, the CMP method as presented here forms the basis for powerful techniques for parameterized verification.

Finally, the CMP method is not just for distributed protocols—though these perhaps are the most challenging and practically relevant examples. It can be used for other types of parameterized systems. In fact, one of the first applications of this method was to verify Tomasulo's Out of Order execution algorithm [52]. There are several possible extensions to this method. Though we have dealt with message passing systems, the same method works well for shared-memory systems such as concurrent software. For instance, in [61] intricate concurrent list data structures are formally verified using the CMP method. Since the CMP method cleanly separates strengthening from abstraction and the only requirement on abstraction is that it be conservative, an interesting research direction would be to try using richer abstractions to further reduce the burden on the user.

21.6 Related Work

Backward reachability analysis based on well quasi-orderings was first introduced in [2].

The RMC framework was first introduced in [45] and then augmented with techniques such as widening [13, 17, 69], abstraction [14–16], and acceleration [5].

A method of particular interest for parameterized systems is that of *counter abstraction*. In general, counter abstraction is designed for systems with unstructured or clique architectures. The idea is to keep track of the number of processes that satisfy a certain property. The technique in [38] generates an abstract system which is essentially a Petri net. Counter-abstracted models with *broadcast communication* are proved to be *well quasi-ordered* in [36]. In [25, 26] symbolic model checking based on real arithmetics is used to verify counter-abstracted models of cache coherence protocols enriched with global conditions. The method works without guarantee of termination. The paper [59] refines the counter abstraction idea by truncating the counters at the value of 2, and thus obtains a finite-state abstract system. The method may require manual insertion of auxiliary program variables for programs that exploit knowledge of process identifiers. In [44] and [68], the authors present a tool for the analysis and verification of *linear parameterized hardware systems* using the *monadic second-order logic on strings*.

Other methods are based on *cut-off* conditions under which parameterized verification can be performed by inspecting only a finite number of system instances. In [33], cut-offs are introduced for token-passing rings, and [29, 31, 32] define cut-offs for systems with disjunctive or conjunctive guards. The paper [4] describes a method, called *view abstraction*, that can be used during the verification procedure in order to dynamically detect cut-off points beyond which the search of the state space need not continue. The *invisible invariants* method [9, 58] exploits cut-off properties to check invariants for mutual exclusion protocols such as the Bakery algorithm and German's protocol. The success of the method depends on the heuristic used in the generation of the candidate invariant. This method sometimes (e.g., for German's protocol) requires insertion of auxiliary program variables for completing the proof. In [10] finite-state abstractions for verification of systems specified in WS1S are computed on the fly by using the weakest precondition operator. The method requires the user to provide a set of predicates on which to compute the abstract model. Heuristics to discover *indexed predicates* are proposed in [48] and applied to German's protocol as well as to the Bakery algorithm. *Environment abstraction* [23] combines predicate abstraction with counter abstraction. The technique is applied to the Bakery and Szymanski algorithms.

Other approaches tailored to snoopy cache protocols modeled with broadcast communication are presented in [32, 51]. In [30] German's directory-based protocol is verified via a manual transformation into a snoopy protocol. In [60] a parameterized version of the Java meta-locking algorithm is verified by means of an induction-based proof technique which requires manual strengthening of the mutual exclusion invariant.

Many induction-based methods were proposed in [18, 21, 45, 47, 50, 62, 70]. In the approaches of [18, 47, 70], the correctness property as well as the inductive invariant are specified as separate processes. In these methods the invariant is specified by the user.

In [21], the authors employ abstract transition systems (ATSs) to specify the invariant. An abstract transition system consists of abstract states and transitions between the abstract states. An abstract state is specified by a regular expression or automaton that denotes a predicate on the global states of systems with an arbitrary number of processes. This approach applies to a large class of networks, i.e., those generated by context free grammars.

In [45], the authors consider parameterized verification of linear and tree networks of processes. They employ regular and tree regular sets for specifying invariance properties of global states and provide an invariant-based method for verification; the invariants are specified symbolically using BDDs and are generated automatically. Their method pertains to verification of invariance properties only.

In [50], a method for verification of families of linear networks is presented. This method uses observer processes as network invariants that are also generated automatically; this method can only be used for verification of safety properties.

In [62] the authors consider families of networks generated by context free grammars. In their approach, the invariant is defined by requiring it to be equivalent to every system of processes generated by the grammar. This is a stronger requirement on the invariant.

In [63], automata on two-dimensional strings have been proposed for expressing correctness properties as well as invariants on computations of parameterized linear networks. An iterative technique for automatically generating the invariant for such families of networks was proposed. This technique can be used for verifying properties under fairness assumptions and hence can be used for both safety and liveness properties. A tool based on the technique was developed and was successfully used for some examples. Fully automated methods have also been proposed in [34] for client-server type architectures. [27] describes a method for automatic verification of ad hoc networks.

Symmetry reduction (see [22, 35, 42]) is an area of research that sometimes employs similar techniques to those employed in parameterized verification. This area of research exploits symmetries in the system to reduce the size of the global state space to be explored. In the model we considered in Sect. 21.2, the principle of "counting the processes in given states" has also been exploited in symmetry reductions. While in symmetry reductions we basically concentrate on verifying a system of fixed size, in parameterized verification we are concerned with verifying systems of all sizes. The relationship between these two areas for other topologies needs further investigation.

Surveys on parameterized verification can be found in [7, 12, 71].

References

1. Abadi, M., Lamport, L.: Composing specifications. In: ACM Transactions on Programming Languages and Systems. ACM, New York (1993)
2. Abdulla, P.A., Cerans, K., Jonsson, B., Tsay, Y.K.: General decidability theorems for infinite-state systems. In: Symp. on Logic in Computer Science (LICS), pp. 313–321. IEEE, Piscataway (1996)

3. Abdulla, P.A., Delzanno, G., Henda, N.B., Rezine, A.: Regular model checking without transducers (on efficient verification of parameterized systems). In: Grumberg, O., Huth, M. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 4424, pp. 721–736. Springer, Heidelberg (2007)
4. Abdulla, P.A., Haziza, F., Hol'ik, L.: All for the price of few (parameterized verification through view abstraction). In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) Intl. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI). LNCS, vol. 7737, pp. 476–495. Springer, Heidelberg (2013)
5. Abdulla, P.A., Jonsson, B., Nilsson, M., d'Orso, J.: Regular model checking made simple and efficient. In: Brim, L., Jancar, P., Kretínský, M., Kucera, A. (eds.) Intl. Conf. on Concurrency Theory (CONCUR). LNCS, vol. 2421, pp. 116–130. Springer, Heidelberg (2002)
6. Abdulla, P.A., Jonsson, B., Nilsson, M., d'Orso, J., Saksena, M.: Regular model checking for LTL(MSO). In: Alur, R., Peled, D.A. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 3114, pp. 348–360. Springer, Heidelberg (2004)
7. Abdulla, P.A., Jonsson, B., Nilsson, M., Saksena, M.: A survey of regular model checking. In: Intl. Conf. on Concurrency Theory (CONCUR). LNCS, vol. 3170, pp. 35–48. Springer, Heidelberg (2004)
8. Apt, K.R., Kozen, D.: Limits for automatic verification of finite-state concurrent systems. Inf. Process. Lett. **22**(6), 307–309 (1986)
9. Arons, T., Pnueli, A., Ruah, S., Xu, J., Zuck, L.D.: Parameterized verification with automatically computed inductive assertions. In: Berry, G., Comon, H., Finkel, A. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 2102, pp. 221–234. Springer, Heidelberg (2001)
10. Baukus, K., Lakhnech, Y., Stahl, K.: Parameterized verification of a cache coherence protocol: safety and liveness. In: Cortesi, A. (ed.) Intl. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI). LNCS, vol. 2294, pp. 317–330. Springer, Heidelberg (2002)
11. Bhattacharya, R., German, S.M., Gopalakrishnan, G.: Exploiting symmetry and transactions for partial order reduction of rule based specifications. In: Valmari, A. (ed.) Intl. Symposium on Model Checking of Software (SPIN). LNCS, vol. 3925. Springer, Heidelberg (2006)
12. Bloem, R., Jacobs, S., Khalimov, A., Konnov, I., Rubin, S., Veith, H., Widder, J.: Decidability of Parameterized Verification. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool, San Rafael (2015)
13. Boigelot, B., Legay, A., Wolper, P.: Iterating transducers in the large (extended abstract). In: Hunt, W.A. Jr., Somenzi, F. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 2725, pp. 223–235. Springer, Heidelberg (2003)
14. Boigelot, B., Legay, A., Wolper, P.: Omega-regular model checking. In: Jensen, K., Podelski, A. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 2988, pp. 561–575. Springer, Heidelberg (2004)
15. Bouajjani, A., Habermehl, P., Rogalewicz, A., Vojnar, T.: Abstract regular (tree) model checking. Int. J. Softw. Tools Technol. Transf. **14**(2), 167–191 (2012)
16. Bouajjani, A., Habermehl, P., Vojnar, T.: Abstract regular model checking. In: Alur, R., Peled, D.A. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 3114, pp. 372–386. Springer, Heidelberg (2004)
17. Bouajjani, A., Touili, T.: Extrapolating tree transformations. In: Brinksma, E., Larsen, K.G. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 2404, pp. 539–554. Springer, Heidelberg (2002)
18. Browne, M., Clarke, E.M., Grumberg, O.: Reasoning about networks with many identical finite state processes. Inf. Comput. **81**(1), 13–31 (1989)
19. Chen, X., Yang, Y., DeLisi, M., Gopalakrishnan, G., Chou, C.T.: Hierarchical cache coherence protocol verification one level at a time through assume guarantee. In: High Level Design Validation and Test Workshop (HLDVT). IEEE, Piscataway (2007)

20. Chou, C.T., Mannava, P.K., Park, S.: A simple method for parameterized verification of cache coherence protocols. In: Hu, A.J., Martin, A.K. (eds.) Formal Methods in Computer Aided Design (FMCAD). LNCS, vol. 3312. Springer, Heidelberg (2004)
21. Clarke, E.M., Grumberg, O., Jha, S.: Verifying parameterized networks using abstraction and regular languages. ACM Trans. Program. Lang. Syst. **19**(5), 726–750 (1997)
22. Clarke, E.M., Jha, S., Enders, R., Filkorn, T.: Exploiting symmetry in temporal logic model checking. Form. Methods Syst. Des. **9**(1/2), 77–104 (1996)
23. Clarke, E.M., Talupur, M., Veith, H.: Environment abstraction for parameterized verification. In: Emerson, E.A., Namjoshi, K.S. (eds.) Intl. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI). LNCS, vol. 3855, pp. 126–141. Springer, Heidelberg (2006)
24. Das, S., Dill, D.L., Park, S.: Experience with predicate abstraction. In: Halbwachs, N., Peled, D.A. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 1633. Springer, Heidelberg (1999)
25. Delzanno, G.: Automatic verification of parameterized cache coherence protocols. In: Emerson, E.A., Sistla, A.P. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 1855, pp. 53–68. Springer, Heidelberg (2000)
26. Delzanno, G.: Verification of consistency protocols via infinite-state symbolic model checking. In: Bolognesi, T., Latella, D. (eds.) Formal Methods for Distributed System Development (FORTE), pp. 171–186. Springer, Heidelberg (2000)
27. Delzanno, G., Sangnier, A., Zavattaro, G.: Parameterized verification of ad hoc networks. In: Gastin, P., Laroussinie, F. (eds.) Intl. Conf. on Concurrency Theory (CONCUR). LNCS, vol. 6269, pp. 313–327. Springer, Heidelberg (2010)
28. Dickson, L.E.: Finiteness of the odd perfect and primitive abundant numbers with n distinct prime factors. Am. J. Math. **35**, 413–422 (1913)
29. Emerson, E.A., Kahlon, V.: Reducing model checking of the many to the few. In: Intl. Conf. on Automated Deduction (CADE). LNCS, vol. 1831, pp. 236–254. Springer, Heidelberg (2000)
30. Emerson, E.A., Kahlon, V.: Exact and efficient verification of parameterized cache coherence protocols. In: Geist, D., Tronci, E. (eds.) Correct Hardware Design and Verification Methods (CHARME). LNCS, vol. 2860, pp. 247–262. Springer, Heidelberg (2003)
31. Emerson, E.A., Kahlon, V.: Model checking guarded protocols. In: Symp. on Logic in Computer Science (LICS), pp. 361–370. IEEE, Piscataway (2003)
32. Emerson, E.A., Kahlon, V.: Rapid parameterized model checking of snoopy cache coherence protocols. In: Garavel, H., Hatcliff, J. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 2619, pp. 144–159. Springer, Heidelberg (2003)
33. Emerson, E.A., Namjoshi, K.S.: Reasoning about rings. In: Cytron, R.K., Lee, P. (eds.) Symp. on Principles of Programming Languages (POPL), pp. 85–94. ACM, New York (1995)
34. Emerson, E.A., Namjoshi, K.S.: Automatic verification of parameterized synchronous systems. In: Alur, R., Henzinger, T.A. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 1102, pp. 87–98. Springer, Heidelberg (1996)
35. Emerson, E.A., Sistla, A.P.: Symmetry and model checking. Form. Methods Syst. Des. **9**(1/2), 105–131 (1996)
36. Esparza, J., Finkel, A., Mayr, R.: On the verification of broadcast protocols. In: Symp. on Logic in Computer Science (LICS), pp. 352–359. IEEE, Piscataway (1999)
37. Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere! Theor. Comput. Syst. **256**(1–2), 63–92 (2001)
38. German, S.M., Sistla, A.P.: Reasoning about systems with many processes. J. ACM **39**(3), 675–735 (1992)
39. Henriksen, J.G., Jensen, J.L., Jørgensen, M.E., Klarlund, N., Paige, R., Rauhe, T., Sandholm, A.: Mona: monadic second-order logic in practice. In: Brinksma, E., Cleaveland, R., Larsen, K.G., Margaria, T., Steffen, B. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 1019, pp. 89–110. Springer, Heidelberg (1995)

40. Higman, G.: Ordering by divisibility in abstract algebras. Proc. London Math. Soc. (3) **2**(7) (1952)
41. Ip, C.N., Dill, D.L.: Better verification through symmetry. In: Proc. Conf. on Computer Hardware Description Languages and Their Applications, pp. 97–111 (1993)
42. Ip, C.N., Dill, D.L.: Better verification through symmetry. Form. Methods Syst. Des. **9**(1/2), 41–75 (1996)
43. Karp, R.M., Miller, R.E.: Parallel program schemata. J. Comput. Syst. Sci. **3**(2), 147–195 (1969)
44. Kelb, P., Margaria, T., Mendler, M., Gsottberger, C.: Mosel: a flexible toolset for monadic second-order logic. In: Brinksma, E. (ed.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 1217, pp. 183–202. Springer, Heidelberg (1997)
45. Kesten, Y., Maler, O., Marcus, M., Pnueli, A., Shahar, E.: Symbolic model checking with rich assertional languages. In: Grumberg, O. (ed.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 1254, pp. 424–435. Springer, Heidelberg (1997)
46. Krstic, S.: Parameterized system verification with guard strengthening and parameter abstraction. In: Automated Verification of Infinite State Systems (2005)
47. Kurshan, R.P., McMillan, K.: A structural induction theorem for processes. In: Rudnicki, P. (ed.) ACM Symp. on Principles of Distributed Computing (PODC), pp. 239–247. ACM, New York (1989)
48. Lahiri, S.K., Bryant, R.E.: Indexed predicate discovery for unbounded system verification. In: Alur, R., Peled, D.A. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 3114, pp. 135–147. Springer, Heidelberg (2004)
49. Lamport, L.: A new solution of Dijkstra's concurrent programming problem. Commun. ACM **17**(8), 453–455 (1974)
50. Lesens, D., Halbwachs, N., Raymond, P.: Automatic verification of parametrized linear networks of processes. In: Boehm, H.-J., Steele, G.L. Jr. (eds.) Symp. on Principles of Programming Languages (POPL), pp. 101–105. ACM, New York (1996)
51. Maidl, M.: A unifying model checking approach for safety properties of parameterized systems. In: Berry, G., Comon, H., Finkel, A. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 2102, pp. 311–323. Springer, Heidelberg (2001)
52. McMillan, K.L.: Verification of an implementation of Tomasulo's algorithm by compositional model checking. In: Hu, A.J., Vardi, M.Y. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 1427. Springer, Heidelberg (1998)
53. McMillan, K.L.: Verification of infinite state systems by compositional model checking. In: Pierre, L., Kropf, T. (eds.) Correct Hardware Design and Verification Methods (CHARME). LNCS, vol. 1703. Springer, Heidelberg (1999)
54. McMillan, K.L.: Parameterized verification of the FLASH cache coherence protocol by compositional model checking. In: Correct Hardware Design and Verification Methods (CHARME). LNCS, vol. 2144, pp. 179–195. Springer, Heidelberg (2001)
55. Misra, J., Chandy, K.M.: Proofs of networks of processes. In: IEEE Transactions on Software Engineering, vol. SE-7. IEEE, Piscataway (1981)
56. O'Leary, J., Talupur, M., Tuttle, M.R.: Protocol Verification using Flows: Parameterized Verification using Message Flows. Formal Methods in Computer Aided Design (FMCAD). IEEE, Piscataway (2009)
57. Park, S., Dill, D.L.: Verification of flash cache coherence protocol by aggregation of distributed transactions. In: SPAA '96: Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures, pp. 288–296. ACM, New York (1996)
58. Pnueli, A., Ruah, S., Zuck, L.D.: Automatic deductive verification with invisible invariants. In: Margaria, T., Yi, W. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 2031, pp. 82–97. Springer, Heidelberg (2001)
59. Pnueli, A., Xu, J., Zuck, L.D.: Liveness with $(0, 1, \infty)$-counter abstraction. In: Brinksma, E., Larsen, K.G. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 2402, pp. 107–122. Springer, Heidelberg (2002)

60. Roychoudhury, A., Ramakrishnan, I.V.: Automated inductive verification of parameterized protocols. In: Berry, G., Comon, H., Finkel, A. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 2102, pp. 25–37. Springer, Heidelberg (2001)
61. Sethi, D., Talupur, M., Schwartz-Narbonne, D., Malik, S.: Parameterized model checking of fine grained concurrency. In: Donaldson, A.F., Parker, D. (eds.) Intl. Symposium on Model Checking of Software (SPIN). LNCS, vol. 7385. Springer, Heidelberg (2012)
62. Shtadler, Z., Grumberg, O.: Network grammars, communication behaviors and automatic verification. In: Sifakis, J. (ed.) Workshop on Automatic Verification Methods for Finite State Systems. LNCS, pp. 151–166. Springer, Heidelberg (1989)
63. Sistla, A.P., Gyuris, V.: Parameterized verification of linear networks using automata as invariants. Form. Asp. Comput. **11**, 402–425 (1999)
64. Suzuki, I.: Proving properties of a ring of finite-state machines. Inf. Process. Lett. **28**(4), 213–214 (1988)
65. Szymanski, B.K.: A simple solution to Lamport's concurrent programming problem with linear wait. In: Lenfant, J. (ed.) International Conference on Supercomputing (ICS), pp. 621–626. ACM, New York (1988)
66. Talupur, M., Krstic, S., O'Leary, J., Tuttle, M.R.: Parametric verification of industrial strength cache coherence protocols. In: Proc. Workshop on Design of Correct Circuits (DCC) (2008)
67. Talupur, M., Tuttle, M.R.: Going with the flow: parameterized verification using message flows. In: Cimatti, A., Jones, R.B. (eds.) Formal Methods in Computer Aided Design (FMCAD). IEEE, Piscataway (2008)
68. Topnik, C., Wilhelm, E., Margaria, T., Steffen, B.: jMosel: a stand-alone tool and jABC plugin for M2L(Str). In: Valmari, A. (ed.) Intl. Symposium on Model Checking of Software (SPIN). LNCS, vol. 3925, pp. 293–298. Springer, Heidelberg (2006)
69. Touili, T.: Regular model checking using widening techniques. Electron. Notes Theor. Comput. Sci. **50**(4), 342–356 (2001)
70. Wolper, P., Lovinfosse, V.: Verifying properties of large sets of processes with network invariants. In: Sifakis, J. (ed.) Workshop on Automatic Verification Methods for Finite State Systems, pp. 68–81. Springer, Heidelberg (1989)
71. Zuck, L.D., Pnueli, A.: Model checking and abstraction to the aid of parameterized systems (a survey). Comput. Lang. Syst. Struct. **30**(3–4), 139–169 (2004)

Chapter 22
Model Checking Security Protocols

David Basin, Cas Cremers, and Catherine Meadows

Abstract The formal analysis of security protocols is a prime example of a domain where model checking has been successfully applied. Although security protocols are typically small, analysis by hand is difficult as a protocol should work even when arbitrarily many runs are interleaved and in the presence of an adversary. Specialized model-checking techniques have been developed that address both the problems of unbounded, interleaved runs and a prolific, highly nondeterministic adversary. These techniques have been implemented in model-checking tools that now scale to protocols of realistic size and can be used to aid protocol design and standardization.

In this chapter, we provide an overview of the main applications of model checking in security protocol analysis. We explain the central concepts involved in the analysis of security protocols: the abstraction of messages, protocols as role automata, the adversary model, and property specification. We explain and relate the main algorithms used and describe systems based on them. We also give examples of the successful applications of model checking to protocol standards. Finally, we provide an outlook on the field: What is possible with the state of the art and what are the future challenges?

22.1 Introduction

Cryptographic protocols are communication protocols that use cryptography to achieve security goals such as secrecy, authentication, and agreement in the presence of adversaries. Examples of well-known cryptographic protocols are SSL/TLS [44], IKEv2 [59], and Kerberos [83], which can be used, respectively, to secure web-

D. Basin
ETH Zürich, Zürich, Switzerland

C. Cremers (✉)
University of Oxford, Oxford, UK
e-mail: cas.cremers@cs.ox.ac.uk

C. Meadows
Naval Research Laboratory, Washington, DC, USA

E.M. Clarke et al. (eds.), *Handbook of Model Checking*,
DOI 10.1007/978-3-319-10575-8_22

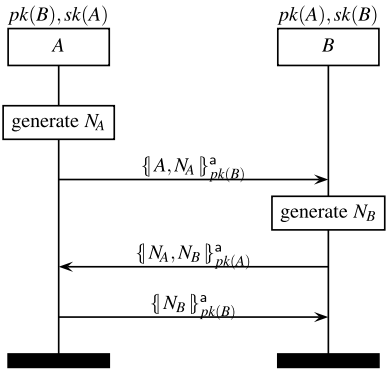


Fig. 1 Needham–Schroeder protocol (NS)

based traffic, set up virtual private networks, and perform authentication in distributed environments. In order to ensure that such protocols always achieve their goals, they are designed under the assumption that the network is completely controlled by an adversary (also called the intruder or attacker). This means that the adversary can intercept, redirect, and alter data, have access to any operation that is available to legitimate agents, and even control one or more legitimate agents and thus access their keys. Given the hostility of the intended environment, it is not surprising that cryptographic protocols are difficult to design and are subject to subtle flaws, even when the cryptographic primitives used, such as encryption and hash functions, are themselves secure.

To give an idea of what can go wrong, consider Lowe's often-cited attack [66] on the Needham–Schroeder public key protocol [82]. The goal of the protocol is to allow two parties to authenticate each other, i.e., after execution of the protocol they can be sure that they have been communicating with the intended partner. The protocol achieves this by combining two challenge–response interactions. Agents can execute the initiator role A or the responder role B. At the end of the protocol, the initiator and the responder agree on a pair of shared secrets, N_A and N_B, where N_A is a random number (or nonce) generated by the initiator and N_B is a nonce generated by the responder. The protocol relies on public-key encryption: anyone can send a message to an agent X using X's public key $pk(X)$, but only X can decrypt it, using its private key $sk(X)$. We write $\{\!|\, M \,|\!\}^{\mathsf{a}}_{pk(X)}$ to denote the (asymmetric) encryption of the message M with X's public key. A message sequence chart describing the protocol is shown in Fig. 1.

Let us go through the protocol steps and their rationale.

1. $A \to B : \{\!|\, A, N_A \,|\!\}^{\mathsf{a}}_{pk(B)}$
   The responder receives the initiator's message and decrypts it. At this point, the responder assumes that the initiator has indeed sent the message recently and will try to confirm his assumption in the next two steps. The responder generates a nonce N_B.
2. $B \to A : \{\!|\, N_A, N_B \,|\!\}^{\mathsf{a}}_{pk(A)}$
   When the initiator receives this message, she decrypts it. Because the message contains the nonce N_A, which the initiator generated recently and sent encrypted

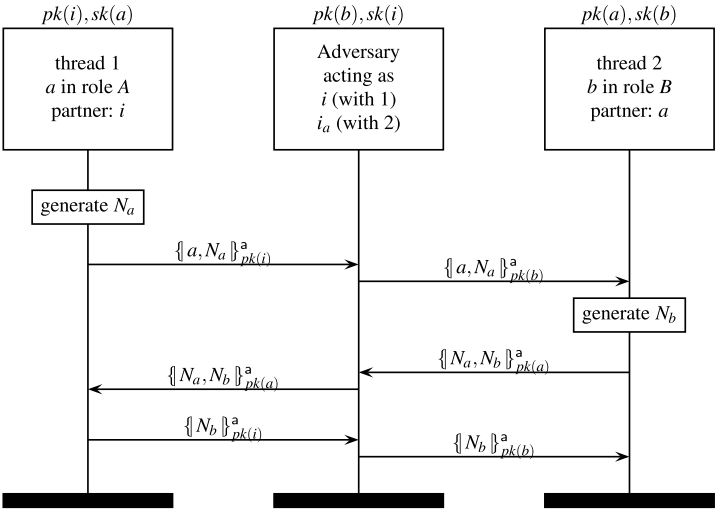


Fig. 2 Lowe's man-in-the-middle attack on the Needham–Schroeder protocol

for the responder, the initiator concludes that the message was indeed sent recently by the responder.

3. $A \to B : \{\!|\, N_B \,|\!\}^{\mathsf{a}}_{pk(B)}$
   The last message is sent so that the responder can verify his assumptions. He reasons that the message is recent, since N_B is recent. Moreover, only the initiator and the responder know N_B, and the responder did not send the message. So the message must come from the initiator. Finally, the initiator will have received N_B as part of the responder's message that also contained N_A, so she would not have responded unless N_A was also recently sent by the initiator.

Although this informal correctness argument may seem convincing, it suffers from the following attack. Here i denotes the adversary and i_a denotes the adversary impersonating agent a. The corresponding message sequence chart is shown in Fig. 2.

The attack proceeds in the following way.

1. $a \to i : \{\!|\, a, N_a \,|\!\}^{\mathsf{a}}_{pk(i)}$
   The agent a initiates the protocol in the initiator role, aiming to communicate with i, and generates a fresh nonce N_a.
2. $i_a \to b : \{\!|\, a, N_a \,|\!\}^{\mathsf{a}}_{pk(b)}$
   The adversary i uses a's nonce to impersonate a and initiates an instance of the protocol with b, who executes the responder role.
3. $b \to a : \{\!|\, N_a, N_b \,|\!\}^{\mathsf{a}}_{pk(a)}$
   b responds to a correctly, generating his own nonce N_b in the process. Since a sees the nonce she sent to i, she assumes the message is from i.
4. $a \to i : \{\!|\, N_b \,|\!\}^{\mathsf{a}}_{pk(i)}$
   a responds to i, following the rules of the initiator role of the protocol.

5. $i_a \rightarrow b : \{\!|\, N_b \,|\!\}^{\mathsf{a}}_{pk(b)}$
   i re-encrypts N_b under b's public key and sends the result to b. Since b is expecting this response from a, he concludes that he shares N_a and N_b with a and a alone.

Since b thinks he is communicating with a, and neither a nor b deviate from the protocol and their keys are not known to the adversary, b expects that N_a and N_b are only known to a and himself. Clearly, b's assumption is violated by the attack.

There are two things to notice about this attack. First, it is nonintuitive: if one looks closely, one sees that the security argument relies on the assumption that agents do not reveal secrets, and that this assumption is violated by i when forwarding a's nonce to b. Indeed, Needham and Schroeder explicitly make this assumption in their paper. However, relaxing the assumption has surprising consequences, as this attack makes clear. Second, the attack does not depend on any flaws in the cryptographic primitives used and requires only a very simple adversary model. The only operations on data that the adversary employs are concatenation and splitting of messages, and encryption and decryption.

In general, we would like to establish the correctness of protocols with respect to even more powerful, but realistic, adversaries. We typically also give the adversary the ability to compute a limited set of functions on data. Namely, the adversary can read, redirect, and delete any message sent along a network, impersonate any agent, and create new messages by applying functions available to him to data he has already seen. This results in a model that can capture a large class of potential protocol problems. Such a model, which is both simple and expressive enough to capture a large class of nonintuitive security flaws, lends itself well to model checking. Hence it is not surprising that the analysis of protocols with respect to such models has been a major application of model checking in security. Basically, one uses the model checker to find all possible ways an adversary can interact with a protocol by using arbitrary combinations of interception, redirection, and the other basic operations available to him. The state space thus generated is of course infinite, but as we will see, the problem is decidable under certain limiting but reasonable restrictions, and heuristics and abstraction techniques have been developed to reason about the case in which the restrictions are not assumed.

Model checking cryptographic protocols is not just of intellectual interest. It can do much to streamline the development and adoption of security standards. New cryptographic protocols are constantly being invented as new communication paradigms are introduced. Since a protocol must be widely adopted before it is useful, new protocols are usually introduced through a standardization process. This process can be drawn-out and argumentative, and standards can be difficult to modify once they are in place. Formal analysis can help speed up standardization by finding problems early and giving evidence of security if no flaws are found. Moreover, it can also help prevent flawed protocols from being standardized. Finally, model checking has the advantage that the counter-examples it finds depict actual attacks on the protocol. This gives insight into a protocol's vulnerabilities, and how they can be fixed.

We proceed as follows. In Sect. 22.2 we give a brief historical overview of the field. Then, in Sect. 22.3, we describe a basic model for security protocols and their properties. In Sect. 22.4 we give an overview of several issues that arise in model checking security protocols, and approaches that have been taken to address them. In Sect. 22.5 we describe representative systems based on these approaches. We discuss current and future research questions in Sect. 22.6 and we draw conclusions in Sect. 22.7.

22.2 History

In this section, we give a brief history of model checking cryptographic protocols.

The first symbolic approach to cryptographic protocol analysis, and the basis of the methods used by current model-checking tools, was that of Dolev and Yao [46, 47], and shortly later, Dolev, Even, and Karp [45], just after the invention of public-key cryptography. They introduced the paradigm now known as the Dolev–Yao model. In this model, a protocol is modeled as a machine consisting of an arbitrary number of honest agents executing the protocol, in which all messages sent are intercepted by the adversary (even if he does no more than forward them), all messages received are sent by the adversary, and any message processing done by the adversary is done using an arbitrary combination of a finite set of operations. The model also formalizes an abstraction of cryptography where messages are represented by terms rather than bit strings and cryptography is "perfect" in the sense that cryptographic operators do not leak information, e.g., the only way for the adversary to decrypt an encrypted message is to have the decryption key.

The Dolev–Yao model is at the basis of all applications of model checking to cryptographic protocols, although today's protocol analysis tools take a very different approach than that taken originally. Dolev and Yao were interested in low-complexity algorithms for proving secrecy, and gave several polynomial-time algorithms for a class of protocols they characterized as "ping-pong" protocols. However, it turned out that the problem quickly became undecidable when the algebraic properties of the cryptographic algorithms were represented more faithfully [52], and interest in the problem petered out.

A few years later, researchers started tackling the problem from another point of view, developing tools that would exhaustively search the problem space or some portion of it. The first tool to take this approach was Millen's Interrogator [78], followed by the Longley–Rigby search tool [65] and the NRL Protocol Analyzer (NPA) [71]. These can be thought of as proto-model checkers. Indeed the NPA offered many features of a model checker, including an automated means of proving that exhaustive search of a finite space implied exhaustive search of the infinite state space, and later, a temporal logic language, NPATRL [96], for describing protocol security properties.

Interest in model checking cryptographic protocols really took off with Lowe's use of the FDR model checker to analyze the Needham–Schroeder public-key protocol [66], described above. The fact that he could demonstrate a problem that had

gone unnoticed for 17 years alerted people to the power of model checking. Other researchers began applying their own model checkers to the problem, most notably the use of the Murphi model checker by Mitchell et al. [80] to analyze variations on the TLS protocol. From there it was a short step to the development of special-purpose model checkers, such as Clarke et al.'s Brutus [35].

Parallel to this was research on the complexity of model checking. The key feature turned out to be the number of sessions involved, where a *session* refers to a single (potentially partial) execution of the protocol. As we see from the attack on the Needham–Schroeder protocol, attacks often interleave different sessions. Indeed it is possible to create protocols that are only vulnerable to attacks that require interleaving an arbitrarily large number of sessions [76]. In [48, 49], Durgin et al. show that, given a model similar to the one described in this chapter, the secrecy problem (that is, the problem of deciding whether or not the adversary learns a particular term) is undecidable if the number of sessions and nonces is unbounded. Rusinowitch and Turuani [88] later showed that the secrecy problem is NP-complete if the number of sessions is bounded. Moreover, their procedure applies more generally to arbitrary state properties, and thus can also be applied to other reachability properties such as authentication.

The decidability of security for the bounded-session case led to the development of *bounded-session* model checkers, in which the user specifies the number of sessions the model checker should search. Bounded-session model checkers are of practical significance, since most attacks on realistic protocols require only a few sessions. Indeed, an attack that requires interleaving many sessions would not be practical to implement. Bounded-session model checkers include Shmatikov and Millen's constraint based tool [79], the Constraint-Logic-based Attack Searcher (CL-Atse) [98], the On-the-Fly Model Checker (OFMC) [20], and the SAT-based Model Checker SAT-MC [5]. The same period saw the development of unbounded-session model checkers relying on abstraction (such as ProVerif [29]) and heuristics (such as Maude-NPA [50] and Athena [94], the latter of which formed the basis for the current tool Scyther [42]). We present these tools in more detail in Sect. 22.5.

In recent years, as the field matures, researchers are increasingly concentrating on making the tools available for others to use, and are applying them to practical problems such as the verification of standards. Some of the tools that have seen the widest use are the AVISPA tool suite [4, 100], which is a set of model checkers (the above-mentioned CL-Atse, SAT-MC, and OFMC), with a common front end, and the above-mentioned ProVerif tool. Many tools have been used in the analysis of standards, sometimes detecting problems that would have gone unnoticed otherwise. For example, the NPA was used in the verification of two IETF protocols: the Internet Key Exchange Protocol [72] and the Group Domain of Interpretation (GDOI) protocol [74]. In the case of GDOI, the tool was instrumental in catching some vulnerabilities early on that were straightforward to fix at the design stage but could have led to problems if they had not been caught in time. The AVISPA tools have been applied to a suite of protocol standards. ProVerif has been used in the production of formally verified implementations of TLS [27] and the smart card protocol InfoCard [28]. Scyther has been used in the analysis of the MQV family

of protocols [16], and has found attacks against members of that family when the adversary is able to compromise parts of the local state of the agents. Furthermore, Scyther has been used for the analysis of the entity authentication protocols in the ISO/IEC 9798 standard [17] and the IKE key exchange protocols in the IPsec standard [43].

Ultimately, we would expect model checking to become a standard tool for cryptographic protocol design, as it has become in hardware design. This has not quite happened yet. Although model checking has proved useful in the analysis of standards, it has not yet become part of the standards designer's basic toolbox. However, designers of new protocols are starting to accompany them with formal analyses. The field is still actively growing and changing, and we would not be surprised to see model checking being more widely adopted in the near future.

Although the basic decidability results and model-checking algorithms for what is commonly accepted as the standard Dolev–Yao model are well understood, there is still much to be learned, and there is currently active research going on in a number of areas. These include making the Dolev–Yao model more precise and expressive, e.g., by including equational properties of cryptographic algorithms such as the associativity-commutativity of exponentiation, and incorporating cryptographic theories of correctness, reasoning about non-trace properties such as non-repudiation, non-interference, and indistinguishability, extending soundness results down to the code level, and handling probabilistic behavior. These topics will be discussed in further detail in Sect. 22.6.

22.3 Formal Model

Each of the tools mentioned in the previous section is based on a formal model of cryptographic protocols and the actions available to the adversary. Although these models differ in their details, they have a number of important features in common, since they are all based on the Dolev–Yao symbolic approach presented in Sect. 22.2. In this section we present a basic symbolic model (based on [16]) for formalizing and reasoning about security protocols, which captures the main features shared by these different models. We will use this model as a reference point when we describe the different tools and approaches later in this chapter.

As the model is symbolic, messages are represented by terms in a term algebra. Protocols themselves are described by a set of roles, each role with an associated script that describes the sequence of events taken by the agents executing the role. The protocols are given an operational semantics where agents may play in multiple roles, giving rise to arbitrarily many role instances (also called threads). This gives rise to a semantics formalized by an infinite-state transition system, with an associated notion of trace.

For expository purposes, we present a simple model that handles only a restricted class of security protocols. We describe these restrictions along with extensions and other design options. We also explain how basic security properties can be formalized within this model.

22.3.1 Notational Preliminaries

Let f be a function. We write $dom(f)$ and $ran(f)$ to denote f's domain and range, respectively. We write $f[a \mapsto b]$ to denote f's update, which is the function f' where $f'(x) = b$ when $x = a$ and $f'(x) = f(x)$ otherwise. We write $f : X \twoheadrightarrow Y$ to denote a partial function mapping some elements from X to elements from Y. We write f^n to denote the n-fold composition of f, for $n \geq 0$, where f^0 is the identity function.

For any set S, $\mathscr{P}(S)$ denotes the power set of S and S^* denotes the set of finite sequences of elements from S. We write $\langle s_0, \ldots, s_n \rangle$ (omitting brackets when no confusion can result) to denote the sequence consisting of the elements s_0 through s_n. For s a sequence of length $|s|$ and $i < |s|$, we write s_i to denote the i-th element. We write $s\hat{}s'$ for the concatenation of the sequences s and s'. Abusing set notation, we write $e \in s$ iff $\exists i \,.\, s_i = e$, and write $set(s)$ for $\{x \mid x \in s\}$.

We use standard notions (see, e.g., [7]) for manipulating terms. Let $\mathscr{S}ub$ denote the set of substitutions of terms for variables (we will define these syntactic categories shortly). We write $[t_0, \ldots, t_n/x_0, \ldots, x_n] \in \mathscr{S}ub$ to denote the substitution of t_i for x_i, for $0 \leq i \leq n$. We extend the functions dom and ran to substitutions. We write $\sigma \cup \sigma'$ to denote the union of two substitutions, which is defined when $dom(\sigma) \cap dom(\sigma') = \emptyset$. We write $\sigma(t)$ for the application of the substitution σ to t.

For R a binary relation, we write R^{-1} to denote the *inverse* of R, i.e., $R^{-1} = \{(y, x) \mid (x, y) \in R\}$. Furthermore, R^+ denotes the transitive closure of R.

22.3.2 Terms and Events

We assume we have the pairwise-disjoint infinite sets *Agent*, *Role*, *Fresh*, *Var*, *Func*, *TID*, and *AdvConst* of agent names, roles, freshly generated terms (nonces, session keys, coin flips, etc.), variables, function names, thread identifiers, and adversary-generated constants.

In order to bind local terms, such as freshly generated terms or local variables, to a protocol role instance (thread), we write $t \sharp tid$. This denotes that the term t is local to the protocol role instance identified by the thread identifier tid, where $tid \in TID$.

Definition 1 Basic terms

$$\begin{aligned} BasicTerm ::= {} & Agent \mid Role \mid Fresh \mid Var \mid AdvConst \\ & \mid Fresh \sharp TID \mid Var \sharp TID \end{aligned}$$

Definition 2 Terms

$$\begin{aligned} Term ::= {} & BasicTerm \mid (Term, Term) \\ & \mid pk(Term) \mid sk(Term) \mid k(Term, Term) \\ & \mid \{\!|\, Term \,|\!\}^{\mathsf{a}}_{Term} \mid \{\!|\, Term \,|\!\}^{\mathsf{s}}_{Term} \mid Func\big(Term^*\big) \end{aligned}$$

For each $X, Y \in Agent$, $sk(X)$ denotes the long-term private key of X, $pk(X)$ denotes the long-term public key belonging to X, and $k(X, Y)$ denotes the long-term symmetric key shared between X and Y. Moreover, $\{\!|\, t_1 \,|\!\}^{\mathsf{a}}_{t_2}$ denotes the asymmetric encryption of the term t_1 with the key t_2, and $\{\!|\, t_1 \,|\!\}^{\mathsf{s}}_{t_2}$ denotes symmetric encryption. Elements of the set *Func* can be used to model other cryptographic functions, such as hash functions. Freshly generated terms and variables are assumed to be local to a thread. We model constants as 0-ary functions.

Depending on the protocol analyzed, we assume that symmetric or asymmetric long-term keys have been distributed prior to protocol execution. We assume the existence of an inverse function on terms, where t^{-1} denotes the inverse key of t. We have $pk(X)^{-1} = sk(X)$, $sk(X)^{-1} = pk(X)$ for all $X \in Agent$, and $t^{-1} = t$ for all other terms t.

As noted in the introduction, one of the distinguishing features of model checking security protocols is that they operate in an environment controlled by an adversary. To formalize the powers of a Dolev–Yao-style adversary, we define a binary relation $\vdash$, where $M \vdash t$ denotes that the term t can be inferred from the set of terms M. Let $t_0, \ldots, t_n \in Term$ and let $f \in Func$. We define $\vdash$ as the smallest relation satisfying:

$$
\begin{aligned}
t \in M &\Rightarrow M \vdash t \\
M \vdash t_1 \wedge M \vdash t_2 &\Leftrightarrow M \vdash (t_1, t_2) \\
M \vdash t_1 \wedge M \vdash t_2 &\Rightarrow M \vdash \{\!|\, t_1 \,|\!\}^{\mathsf{s}}_{t_2} \\
M \vdash t_1 \wedge M \vdash t_2 &\Rightarrow M \vdash \{\!|\, t_1 \,|\!\}^{\mathsf{a}}_{t_2} \\
M \vdash \{\!|\, t_1 \,|\!\}^{\mathsf{s}}_{t_2} \wedge M \vdash t_2 &\Rightarrow M \vdash t_1 \\
M \vdash \{\!|\, t_1 \,|\!\}^{\mathsf{a}}_{t_2} \wedge M \vdash (t_2)^{-1} &\Rightarrow M \vdash t_1 \\
\bigwedge_{0 \le i \le n} M \vdash t_i &\Rightarrow M \vdash f(t_0, \ldots, t_n)
\end{aligned}
$$

Subterms t of a term t', written $t \sqsubseteq t'$, are defined as the syntactic subterms of t', e.g., $t_1 \sqsubseteq \{\!|\, t_1 \,|\!\}^{\mathsf{s}}_{t_2}$ and $t_2 \sqsubseteq \{\!|\, t_1 \,|\!\}^{\mathsf{s}}_{t_2}$. We write $FV(t)$ for the free variables of t, defined as $FV(t) = \{t' \mid t' \sqsubseteq t \wedge t' \in Var \cup \{v \sharp tid \mid v \in Var \wedge tid \in TID\}\}$.

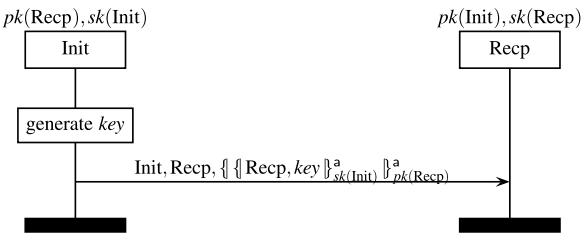
An agent can engage in the following events.

Definition 3 Events

$$Event ::= \mathsf{create}(Role, \mathscr{S}ub) \mid \mathsf{send}(Term) \mid \mathsf{recv}(Term)$$

Note that the send and receive events do not include explicit sender or recipient fields. The messages sent or received can, of course, include subterms identifying the sender and the intended recipient, although this information is not a priori authentic. As is standard, the adversary receives all messages sent, independent of the intended recipient.

Fig. 3 Simple protocol



We will explain the interpretation of the events shortly. Here we note that the events are conventional and are given a standard interpretation in a setting with concurrently executing, communicating processes: starting a thread, sending a message, and receiving a message.

We extend the domain of substitutions over events and sequences of events in the standard way, i.e., $\sigma(\mathsf{send}(m)) = \mathsf{send}(\sigma(m))$.

22.3.3 Protocols and Threads

A protocol is a mapping from role names to event sequences, i.e., $Protocol : Role \twoheadrightarrow Event^*$. We require that no thread identifiers occur as subterms of events in a protocol definition. The following is a very simple example of a protocol with two roles: an initiator and a recipient.

Example 1 (Simple protocol) Let $\{\text{Init}, \text{Recp}\} \subseteq Role$, $key \in Fresh$, and $x \in Var$. Let P be the protocol defined as follows.

$$P(\text{Init}) = \langle \mathsf{send}(\text{Init}, \text{Recp}, \{\!|\, \{\!|\, \text{Recp}, key \,|\!\}^{\mathsf{a}}_{sk(\text{Init})} \,|\!\}^{\mathsf{a}}_{pk(\text{Recp})}) \rangle$$

$$P(\text{Recp}) = \langle \mathsf{recv}(\text{Init}, \text{Recp}, \{\!|\, \{\!|\, \text{Recp}, x \,|\!\}^{\mathsf{a}}_{sk(\text{Init})} \,|\!\}^{\mathsf{a}}_{pk(\text{Recp})}) \rangle$$

The message sequence chart for this protocol is shown in Fig. 3. Here, the initiator generates a key and sends it (together with the recipient's name) signed and encrypted, along with the initiator and recipient names. The recipient expects to receive a message of this form.

Protocols are executed by agents who execute roles, thereby instantiating role names with agent names. Agents may execute each role multiple times. We distinguish between the fresh terms and variables of each thread by assigning them unique names, using the function $localize : TID \to \mathscr{S}ub$. Note that we abuse notation and extend the domain of substitutions to $Var \cup Role \cup Fresh$.

Definition 4 (Localize) Let $tid \in TID$. Then

$$localize(tid) = \bigcup_{cv \in Fresh \cup Var} [cv \sharp tid / cv].$$

We define a function $thread : (Event^* \times TID \times \mathscr{S}ub) \to Event^*$ that yields the sequence of agent events that may occur in a thread.

Definition 5 (Thread) Let l be a sequence of events, $tid \in TID$, and let σ be a substitution. Then

$$thread(l, tid, \sigma) = \sigma\big(localize(tid)(l)\big).$$

Example 2 Let P be the protocol from Example 1, $t_1 \in TID$, and $\{A, B\} \subseteq Agent$. For a thread t_1 performing the Init role we have $localize(t_1)(key) = key\sharp t_1$ and

$$\begin{aligned} &thread\big(P(\text{Init}), t_1, [A, B/\text{Init}, \text{Recp}]\big) \\ &= \big\langle \mathsf{send}\big(A, B, \{\!|\, |\!\}^{\mathsf{a}}_{sk(A)} \{\!|\, B, key\sharp t_1 \,|\!\}^{\mathsf{a}}_{pk(B)}\big)\big\rangle. \end{aligned}$$

22.3.4 Initial Adversary Knowledge

We assume that the adversary initially knows all agent names and can generate an unbounded set of constants *AdvConst*, where $AdvConst \subset Func$ and no protocol description contains elements of *AdvConst*. The set *AdvConst* represents the set of fresh values that are generated by the adversary. The adversary additionally knows the long-term public keys of all agents. We also assume that the adversary has compromised the long-term private keys of some of the agents. We model this by partitioning the set *Agent* into the honest agent set *Honest* and the compromised agent set *Compromised*. Moreover, we include the long-term private keys of the compromised agents in the initial knowledge of the adversary.

The long-term secret keys of an agent a are defined as

$$LongTermKeys(a) = \big\{sk(a)\big\} \cup \bigcup_{b \in Agent} \big\{k(a, b), k(b, a)\big\}.$$

We define the initial adversary knowledge IK_0 as

$$IK_0 = Agent \cup AdvConst \cup \bigcup_{a \in Agent} \big\{pk(a)\big\} \cup \bigcup_{a \in Compromised} LongTermKeys(a).$$

22.3.5 Execution Model

We define the set *Trace* as $(TID \times Event)^*$, which represents possible execution histories. Using this set, we define the set *State* of system states as $Trace \times \mathscr{P}(Term) \times (TID \nrightarrow Event^*)$. The components of a state $(tr, IK, th) \in State$ are (1) a trace tr, (2) the adversary knowledge IK, and (3) a partial function th mapping thread identifiers

$$\frac{R \in dom(P) \quad \sigma \in dom(P) \to Agent \quad tid \notin dom(th)}{(tr, IK, th) \longrightarrow (tr^\frown\langle(tid, \mathsf{create}(R, \sigma))\rangle, IK, th[tid \mapsto thread(P(R), tid, \sigma)])}\,[\mathsf{create}_P]$$

$$\frac{th(tid) = \langle \mathsf{send}(m)\rangle^\frown l}{(tr, IK, th) \longrightarrow (tr^\frown\langle(tid, \mathsf{send}(m))\rangle, IK \cup \{m\}, th[tid \mapsto l])}\,[\mathsf{send}]$$

$$\frac{th(tid) = \langle \mathsf{recv}(pt)\rangle^\frown l \qquad IK \vdash \sigma(pt) \qquad dom(\sigma) = FV(pt)}{(tr, IK, th) \longrightarrow (tr^\frown\langle(tid, \mathsf{recv}(\sigma(pt)))\rangle, IK, th[tid \mapsto \sigma(l)])}\,[\mathsf{recv}]$$

Fig. 4 Execution-model rules

of initiated threads (executing or completed) to event traces. Note that in conventional model-checking approaches, (1) would not be part of the state but would be defined over runs of the transition system. We include the trace as part of the state to facilitate defining the partner function later.[1]

The initial system state s_{init} is defined as

$$s_{\text{init}} = \left(\langle\rangle, IK_0, \emptyset\right),$$

where IK_0 is the initial adversary knowledge as defined above.

The semantics of a protocol $P \in Protocol$ is defined by a transition system whose transitions are given by the rules in Fig. 4. Each rule describes how the execution of one of the three events causes a state transition. We describe each rule in turn.

Execution-model rules. The create rule starts a new instance of a protocol role R (a *thread*). In the premises, the substitution σ associates the role names $dom(P)$ with agents, and tid is a fresh thread identifier. In the state transition in the conclusion, the successor state's trace is extended, reflecting that the thread identified by tid executed the create event, and the thread mapping ("thread pool") is extended with the thread assigned to tid.

The send rule sends a message m to the network. The premise refers to a thread identified by tid, whose next event is to send the message m. In the conclusion, the trace is updated with this event, the adversary knowledge is updated with m, and thread for tid is updated.

Finally, the receive rule models an agent, running a thread, receiving a message from the network. The message must match the message pattern pt under a substitution σ, where pt is a term that may contain free variables. Note that by the second premise, the adversary must be able to infer $\sigma(pt)$ from his current knowledge IK. One can see this as formalizing that the adversary controls the network and effectively determines who receives which messages. In our model, recipients accept all messages that match the message pattern pt, and block on any other messages. The resulting substitution σ is applied to the remaining protocol steps l.

Definition 6 (Transition relation) Let P be a protocol. We define a transition relation $\to_P$ using the execution-model rules from Fig. 4. For states s and s', $s \to_P s'$

[1] Actually there are a number of representation options here. For example, IK and th can be computed directly from tr. We explicitly include them in the state to improve the readability of our operational semantics.

iff there exists an execution-model rule with the premises $Q_1(s), \ldots, Q_n(s)$ and the conclusion $s \to s'$ such that all of the premises hold.

Given a protocol P and a set of states T, let Post_P and Pre_P denote the successors and predecessors of T, respectively, i.e.,

$$\mathrm{Post}_P(T) = \left\{ s' \in \mathit{State} \;\middle|\; \exists s \in T \,.\, s \to_P s' \right\}$$
$$\mathrm{Pre}_P(T) = \left\{ s \in \mathit{State} \;\middle|\; \exists s' \in T \,.\, s \to_P s' \right\}.$$

Definition 7 (Reachable states) Let P be a protocol. We define the set of reachable states of P as

$$\mathit{Reachable}(P) = \bigcup_{n=0}^{\infty} \mathrm{Post}_P^n\left(\{s_{\mathrm{init}}\}\right).$$

We will see Pre and Post again when we discuss model-checking algorithms in Sect. 22.4.

22.3.6 Property Specification

We focus on basic security properties that can be expressed as reachability properties, i.e., properties of reachable states. Because the adversary knows the long-term private keys of the compromised agents, protocol sessions that involve compromised agents cannot guarantee the secrecy of data such as shared keys or exchanged terms. This is reflected in the definition of most security properties by considering only those threads that do not involve compromised agents.

We introduce an auxiliary predicate HT (for *honest thread*) that identifies completed threads that do not involve compromised agents.

Definition 8 (HT) Let $s = (tr, IK, th)$ be a state, tid a thread identifier, and σ a substitution. We write $\mathrm{HT}(s, tid, \sigma)$ to denote

$$\exists R \,.\, \big(tid, \mathsf{create}(R, \sigma)\big) \in tr \wedge th(tid) = \langle\rangle \wedge \mathit{ran}(\sigma) \cap \mathit{Compromised} = \emptyset \,.$$

For example, let $s = (tr, IK, th)$ be the state reached after the attack trace represented in Fig. 2. We have that both threads 1 and 2 are completed, i.e. $th(1) = th(2) = \langle\rangle$, and the trace tr contains $(1, \mathsf{create}(A, [a, i/A, B]))$ and $(2, \mathsf{create}(B, [a, b/A, B]))$. Hence we have that $\mathrm{HT}(s, 2, [a, b/A, B])$ but there exists no σ such that $\mathrm{HT}(s, 1, \sigma)$, because a in thread 1 starts a thread to communicate with the compromised agent i.

Definition 9 (Secrecy) Let $t \in \mathit{Fresh}$. We say that a state $s = (tr, IK, th)$ *satisfies secrecy of* t if and only if

$$\forall tid, \sigma \,.\, \mathrm{HT}(s, tid, \sigma) \Rightarrow \neg\big(IK \vdash (t \sharp tid)\big) \,.$$

We say that a protocol P *ensures secrecy of* t if and only all reachable states of P satisfy secrecy of t.

For example, the protocol in Fig. 3 ensures secrecy of the initiator's key. In contrast, the Needham–Schroeder protocol from Fig. 1 does not ensure secrecy of the nonces.

Authentication is an important property for many security protocols, and numerous notions of authentication have been proposed in the literature. As a simple example, consider *weak aliveness*. This weak form of authentication guarantees only that if a non-compromised agent completes a thread of the protocol under the assumption that he is communicating with a non-compromised agent a executing role R, then it is indeed the case that a previously started a thread in role R.

Definition 10 (Weak Aliveness) Let R be a role. We say a state $s = (tr, IK, th)$ *satisfies weak aliveness of* R if and only if

$$\forall tid', \sigma' \,.\, \mathrm{HT}\big(s, tid', \sigma'\big) \Rightarrow \exists tid, \sigma \,.\, \big(tid, \mathsf{create}(R, \sigma)\big) \in tr \wedge \sigma'(R) = \sigma(R) \,.$$

We say that a protocol P *ensures weak aliveness* if and only if all reachable states of P satisfy weak aliveness of all roles $R \in dom(P)$.

For example, the Needham–Schroeder protocol from Fig. 1 satisfies weak aliveness.

Stronger forms of authentication (see, e.g., [67]) impose additional requirements on the state. For example, they require that the threads' assumptions on agents match, e.g., by requiring $\sigma = \sigma'$, or they place additional requirements on the exchanged messages or the instantiation of variables, or they require that messages are recent. Some of these properties require instrumenting the model with additional markers, such as labeling communications or introducing signal events to simplify expressing agreement on the contents of variables.

22.3.7 Alternatives

We have intentionally kept our formalism simple to highlight the main ideas. At each step along the way there are design options, reflecting the class of protocols and adversaries one intends to capture.

To begin with, we have formalized cryptographic messages using a free term algebra, where term equality is therefore just syntactic equality. Additional cryptographic operators can be added, but this requires formalizing how the adversary can construct and reason about terms built from them. In Sect. 22.6 we describe how this can be done for operators formalized by sets of equations.

Our protocol roles are specified by straight-line sequences of events, without control flow primitives such as branching or loops. This is sufficient to model many security protocols, provided we ignore error cases, for example where a thread receives an unexpected message. Such cases are handled implicitly: no transition is

enabled and hence the thread simply does not progress. In contrast, a richer execution model would be needed to fully model protocols that support multiple options and subprotocols, such as the Internet Key Exchange (IKE) [54], or that require loops, such as the stream authentication protocol TESLA [57, 86]. It is not difficult to add control flow primitives or alternatively to base the execution model on a process calculus or some transition-system formalism. We will give examples of tools whose input languages are based on such formalisms in Sect. 22.5.

The simplicity and power of the Dolev–Yao adversary model has made it extremely popular. However, for many real-world scenarios an adversary who has complete control of the network may be unrealistic, and therefore protocols that offer weaker security guarantees may be preferred for efficiency reasons. At the other end of the spectrum, assuming that the adversary can learn nothing about encrypted data unless he obtains a decryption key may be unrealistic in some scenarios. We will consider alternative adversary models in Sect. 22.6.

22.4 Issues in Developing Model-Checking Algorithms for Security Protocols

Here we present a number of issues that arise in model checking security protocols, and the approaches that have been taken to address them. In particular, we indicate various design decisions, such as forward or backward search, state representations, and bounding the state space.

22.4.1 Forward Versus Backward Search

We consider security properties that can be expressed as reachability properties, i.e., as a set of states S. We say that a protocol P satisfies the property S if and only if

$$Reachable(P) \subseteq S. \tag{1}$$

Let $\overline{S} = State \setminus S$ be the property's complement, representing possible attacks. For example, for the secrecy of a term t as in Definition 9, $\overline{S}$ is defined as:

$$\left\{ s \in State \;\middle|\; \exists tid, \sigma \,.\, \mathrm{HT}(s, tid, \sigma) \wedge IK \vdash (t \sharp tid) \right\}.$$

Using the complement $\overline{S}$, Formula (1) can be rewritten as

$$Reachable(P) \cap \overline{S} = \emptyset. \tag{2}$$

Then, we can rewrite Formula (2) either as

$$\left(\bigcup_{n=0}^{\infty} \mathrm{Post}_P^n \left(\{ s_{\mathrm{init}} \} \right) \right) \cap \overline{S} = \emptyset \tag{3}$$

or alternatively as

$$s_{\text{init}} \notin \bigcup_{n=0}^{\infty} \text{Pre}_P^n(\overline{S}). \tag{4}$$

Algorithms that iteratively compute a representation of (all, or some subset of) $\text{Post}_P^n(\{s_{\text{init}}\})$, as in Formula (3), are said to use *forward search*. If, for some n, an element is found that is also in $\overline{S}$, a counterexample can be constructed representing an attack. Alternatively, if there is an n where we reach a fixpoint, i.e. $\text{Post}_P^n(\{s_{\text{init}}\}) = \text{Post}_P^{n+1}(\{s_{\text{init}}\})$, and additionally $\text{Post}_P^n(\{s_{\text{init}}\}) \cap \overline{S} = \emptyset$, then the property holds of the protocol. A fixpoint will be reached, in general, only in finite-state models.

In contrast, *backward search* iteratively constructs $\text{Pre}_P^n(\overline{S})$, as in Formula (4). Similar observations hold as for forward search, except that we check whether s_{init} occurs in the constructed set.

In the analysis of security protocols, the set of reachable states is infinite, as new threads can always be created. Hence the closure in forward search contains infinitely many states. Similarly, the closure in backward search contains infinitely many states, but for a different reason: for the properties we consider here, $\overline{S}$ contains infinitely many states.

The main idea behind searching infinite sets of states is to use finite representations of the infinite sets. The selection criteria for such a finite representation include the complexity of computing Pre_P or Post_P, and the complexity of evaluating whether or not all elements of the represented set satisfy the property S.

When exploring infinite state spaces, it is often efficient to use as much information as possible about the states. In general, the negation $\overline{S}$ of the security property provides more information about the states than the initial state $\{s_{\text{init}}\}$. For example, the negation will specify that particular events must have occurred or that the adversary knows certain terms. As a result, backward search is often employed when exploring infinite sets of states.

A simpler case occurs if the number of reachable states is restricted to a finite set, for example, by limiting the number of threads or sessions that can be created. In this case, forward search for violations of secrecy or authentication properties can be trivially implemented: checking that a given state satisfies these properties can be done using either Definition 9 or 10. A bounded backward search starts from the finite set of attack states $\overline{S}$, from which $\text{Pre}_P^n(\overline{S})$ can be computed. Depending on the property and the (finite) size of the set of states, the size of (the representation of) $\overline{S}$ can be significant. In practice, forward search is commonly used to explore finite sets of states.

22.4.2 Bounded Instances

The execution model presented in Sect. 22.3 gives rise to an infinite-state transition system. Infinitely many states arise in two distinct ways. First, the create rule

may start unboundedly many new protocol threads. Second, under the receive rule, there are unboundedly many different messages that a thread could receive from the network. This is modeled by the rule's second premise, which formalizes that a thread can be updated with any message $\sigma(pt)$ that is in the closure of the adversary knowledge.

The first source of infinity is a fundamental problem. As mentioned in Sect. 22.2, even relatively simple properties such as secrecy are undecidable for security protocols formalized using operational semantics similar to ours [48]. If we restrict the number of sessions (or threads), then the problem becomes NP-complete [88], provided messages are formulated as terms in a free term algebra. Note that in practice, when analyzing real-world protocols, it is usually only necessary to consider a small number of threads. If there is an attack on the protocol, then there is normally an attack where the number of threads is at most a small factor more than the number of roles, e.g., twice the number of roles, which allows for messages from one protocol session to be replayed in another session. For a class of protocols where attacks require arbitrarily many threads, see [76].

The second source of infinity, an infinite space of messages, turns out not to be a problem. The NP-completeness result of Rusinowitch and Turuani [88] establishes that if there is an attack, then there is one where the size of the messages involved is polynomially bounded in the size of the protocol and the number of threads, provided messages are represented by directed acyclic graphs. As a result, assuming a finite number of threads, and hence fresh data, one can bound a priori the messages that must be considered in protocol analysis. We will see below how both bounds on the number of threads and messages have been used by different protocol analysis tools.

22.4.3 Representing States

Formulas (3) and (4) can be directly used for forward or backward model checking after fixing a representation of states for which one can effectively compute successors or predecessors. There are different options here depending on whether states are explicitly or symbolically represented.

It is simple to turn the operational semantics given in Sect. 22.3 into an explicit-state model checker. As defined in Sect. 22.3.5, a state is just a triple, all of whose components can be finitely encoded. The problem in practice is efficiently representing large sets of states, i.e., reducing the impact of state-space explosion. An example of a model checker for security protocols based on forward search using explicit-state enumeration is Murphi [95]. This tool uses techniques inspired by explicit-state model checkers like SPIN [56], such as hash tables and hash compaction, to improve its efficiency. Another example of an explicit-state coding is implemented by Lowe's Casper system [68], which encodes the operational semantics of the protocol and the adversary as a (finite-state) CSP process and uses the FDR model checker to either identify attacks or verify the protocol for instances with a bounded number of threads and messages.

The second possibility is to encode sets of states using formulas as in symbolic model checking. Many tools take this approach. Here terms, in particular messages, are represented by non-ground terms (cf. *message patterns* in Sect. 22.3.5) which contain variables. These variables may be instantiated during search. For example, under our operational semantics, this instantiation would occur when applying the rules using unification. In approaches based on rewriting, such as Maude-NPA, instantiation occurs during rewriting by narrowing.

When working with symbolic representations, unification is often combined with constraint solving. In the formal model we have given, the need for constraint solving arises from the second premise of the recv rule, $IK \vdash \sigma(pt)$. When applying rules of the operational semantics backwards (either in forward or backward search), this gives rise to a subgoal often called the *intruder deduction problem*. The simplest version of this problem is to determine whether $IK \vdash m$, for a ground message m using the rules formalizing the Dolev–Yao adversary given in Sect. 22.3.2. This problem is often decidable, which can be shown by using the notion of locality [18, 38] to bound the size of terms occurring in derivations. During symbolic reasoning, the non-ground problem arises: determining whether there exists a substitution σ such that $\sigma(IK) \vdash \sigma(pt)$. The ground problem can be tackled by considering a restricted class of normal form derivations [38]. The non-ground problem is generally solved either by unification-based procedures or specialized constraint solvers such as those used in OFMC or CL-Atse.

An alternative symbolic approach is that of bounded model checking. In the simplest case, the closure specified in Formula (3) is simply unrolled some bounded number of times k, thereby specifying the existence of an attack given by k or fewer applications of rules from the operational semantics. If this finite unrolling is combined with a bound on the number of messages that may appear (and the result of Rusinowitch and Turuani gives us an exponential bound), then the resulting formula can be encoded within propositional logic and SAT solvers can be used to search for attacks, as shown in Chap. 9 of this handbook [69]. Different encodings and optimizations for using SAT-based model checking for security protocols have been explored by Armando and Compagna and implemented in the model checker SAT-MC [6].

Symbolic representations of terms can be combined with partially ordered finite sets of events to represent (possibly infinite) sets of states or traces. Such a partially ordered set E is used to represent all traces of the protocol that contain an instance of E as a substructure. By applying the operational semantics backwards to the events in E, additional constraints on its traces can be derived from E, such as adding preceding events, unifying messages, or adding constraints on the adversary knowledge. This process can either lead to a contradiction, in which case E represents no traces of the protocol, or to a witness trace of the protocol that contains an instance of E. Such representations form the basis of Athena, Scyther, and Tamarin.

22.4.4 Partial-Order Reduction

Partial-order reduction, as discussed in Chap. 6 of this handbook [85], is a natural optimization technique in the context of model checking security protocols. The reasons for this are twofold. First, separate threads are largely independent processes, which communicate only through a single shared channel. Second, secrecy (Definition 9) depends only on the adversary knowledge and, in our definition, on the communication partners of completed threads.

Example 3 (POR) As a simple example, consider a state s with two threads identified by tid_1 and tid_2. Assume that the next actions of both threads are respectively the receive events e_1 and e_2 of the patterns pt_1 and pt_2, and that there exist messages in the adversary knowledge such that both e_1 and e_2 can be executed. More formally, consider the state $s = (tr, IK, th)$, two sequences l_1, l_2, and two substitutions σ_1, σ_2, such that for all $i \in \{1, 2\}$,

$$e_i = \mathsf{recv}(pt_i) \wedge th(tid_i) = \langle e_i \rangle^\frown l_i \wedge IK \vdash \sigma_i(pt_i) \wedge dom(\sigma_i) = FV(pt_i),$$

and where $FV(pt_1) \cap FV(pt_2) = \emptyset$.

Observe that in this state, e_1 and e_2 can be executed in any order. This results in either s_1 or s_2, where

$$s \to_P^* s_1, \quad \text{and} \quad s_1 = \big(tr^\frown\langle(tid_1, \sigma_1(e_1))\rangle^\frown\langle(tid_2, \sigma_2(e_2))\rangle, IK, th'\big),$$
$$s \to_P^* s_2, \quad \text{and} \quad s_2 = \big(tr^\frown\langle(tid_2, \sigma_2(e_2))\rangle^\frown\langle(tid_1, \sigma_1(e_1))\rangle, IK, th'\big),$$

and where $th' = th[tid_1 \mapsto \sigma_1(l_1)][tid_2 \mapsto \sigma_2(l_2)]$. Observe that s_1 is identical to s_2 except for its trace component. Because the premises of the transition rules do not depend on a state's trace component, the successor states of s_1 are identical to those of s_2 except for the traces.

To simplify the example, we additionally assume that for $i \in \{1, 2\}$,

$$\forall tid, \sigma \,.\, \mathrm{HT}(s, tid, \sigma) \Leftrightarrow \mathrm{HT}(s_i, tid, \sigma).$$

Hence, if the secrecy property is violated in state s_1, then it is also violated in s_2, and vice versa. Thus, we can safely explore only one of these successor states: if there is a state reachable from s_1 that violates secrecy, then there will also be a state reachable from s_2 that is identical up to the trace component in which secrecy is violated, and vice versa.

Similarly, for secrecy properties, one can consider only paths in which threads with send actions are executed first (and ignore paths in which these same sends are executed later) as this will only provide the adversary with more knowledge earlier.

For authentication properties, POR techniques are not necessarily sound: authentication properties depend on the order in which events occur, and therefore ignoring some orderings may cause attacks to be missed. For each property (or class of

properties), the soundness of a particular partial-order reduction scheme must be individually proven. The main proof obligation is to show that if there is an attack in a state that is not explored by the POR scheme, then there is also an attack on a state that is explored in the scheme.

POR techniques are used in several tools. Examples include Brutus [35], OFMC [20], and Maude-NPA [50].

22.4.5 Handling Equations

The formal model introduced in Sect. 22.3 uses a free algebra to represent operations on data. Thus the adversary's ability to perform encryption is represented by the deduction rule stating that $\{\!|\, t_1 \,|\!\}^{\mathsf{a}}_{t_2}$ may be deduced from t_1 and t_2, and his ability to perform decryption is represented by the deduction rule stating that t_1 may be deduced from $\{\!|\, t_1 \,|\!\}^{\mathsf{a}}_{t_2}$ and t_2^{-1}. This approach is in general adequate for a large class of operators, but we run into trouble when we include operations such as exclusive-or which obey different equational theories. In this case, we need to include not only a rule such as

$$M \vdash t_1 \wedge M \vdash t_2 \Rightarrow M \vdash t_1 \oplus t_2$$

but also the set of equations

$$\begin{aligned} t \oplus 0 &= t & t_1 \oplus t_2 &= t_2 \oplus t_1 \\ t \oplus t &= 0 & t_1 \oplus (t_2 \oplus t_3) &= (t_1 \oplus t_2) \oplus t_3 \end{aligned}$$

Similar problems arise when we introduce protocols based on Diffie–Hellman exponentiation or protocols based on homomorphic encryption, such as those involving blind signatures.

There are two ways of dealing with this problem. The first is to replace the equational theory with a set of inference rules that is equivalent under certain syntactic restrictions on the protocol. This is the approach followed in the previously mentioned work on encryption–decryption [77] and also by Küsters and Truderung, who develop inference rules for the theory governing exclusive-or [61] and a subtheory of the theory governing Diffie–Hellman exponentiation [62]. This approach is most appropriate when one wishes to use a tool that does not directly support reasoning about equational properties.

The second approach is to adapt the reasoning used by the tool to the equational theory at hand. A substantial amount of work has been done in this area. Researchers have concentrated on two main techniques. One is an extension of the intruder deduction problem to include equational theories. Thus, we now ask: given a set of terms M, a term t, and an equational theory E, is it possible to determine whether or not $M \vdash t$ modulo E? The decidability of the intruder deduction problem in the free theory was the main component of Rusinowitch and Turuani's proof of decidability

of security in the bounded-session model. Decision procedures have subsequently been given for a large class of equational theories relevant for model checking cryptographic protocols [1]. These theories include a class of rewrite theories known as subterm-convergent (for which the intruder deduction problem is decidable in polynomial time), and other theories such as homomorphic encryption and exclusive-or. Algorithms for verifying intruder deduction modulo finite convergent rewrite theories have been implemented in several tools [23, 34, 39].

The other technique is equational unification: given an equational theory E and two terms t and s, find a complete description of all the substitutions σ such that $\sigma(s) = \sigma(t)$ modulo E. This is useful for determining which states can immediately follow (or precede) a given state: one unifies the current state with the output (or input) of a state transition. Equational unification was a well-known technique long before it was applied to cryptographic protocol analysis. Indeed, anyone who has solved an arithmetic equation such as $x + 7 = 12 + y$ has applied equational unification. However, unification research generally concentrated on algorithms for special-purpose theories. In cryptographic protocol analysis, it is necessary to be able to apply unification to a range of theories, and to different combinations of theories. Thus generic approaches that apply to classes of theories that can be easily combined are preferable to special purpose algorithms, even when the special purpose algorithms may be more efficient. One technique that follows this approach is the process known as *variant narrowing* [51], which can be applied to a class of theories that satisfy a property known as the *finite variant property* [37]. This is satisfied by a large number of equational theories of interest for cryptographic protocol analysis, with the major exception being homomorphic encryption. A version of variant narrowing has been implemented in the Maude-NPA protocol analysis tool, discussed in Sect. 22.5.1.

We note that although there is a large overlap between subterm-convergent and finite variant theories, the finite variant property does not imply subterm convergence, since by definition no associative-commutative theories are subterm convergent. Whether the converse holds is, to the best of our knowledge, still an open problem.

The most pressing open problem in the short term is how to extend the available tools to handle additional theories. In some cases (e.g., the intruder deduction problem for non-subterm-convergent theories) the exact complexity is not known and needs to be understood better. In other cases (for example some of the unification problems connected with different homomorphic encryption theories), the problem is known to be intractable or undecidable, and what instead needs to be investigated are (preferably syntactic) conditions on cryptographic protocol specifications that make the problems tractable. There is also the issue of combining theories. For example, in the case of unification, general algorithms for combining unification algorithms for different theories have been known for some time [91], but their generality forces them to be highly nondeterministic, and thus they are usually too slow to be practical. It may be possible to restrict ourselves to particular types of theories such that practical algorithms can be developed. However, it is still unknown whether such classes of theories contain the main theories of interest for cryptographic protocol analysis.

More generally, there is the problem of determining what one actually learns from an analysis with respect to an equational theory. For example, Kremer et al. [60] have been using subterm-convergent theories to analyze voting protocols. However, the primitives that they describe with subterm-convergent theories are implemented by algorithms satisfying richer equational theories, usually involving Abelian groups. Are the higher-level descriptions in terms of subterm-convergent theories safe approximations? Conversely, when would a free theory be a safe approximation for a subterm-convergent theory? This question has been answered for the subterm-convergent theories involving both shared and public key encryption [77]. Can this result be extended to more general cases? Finally, when is any theory a safe approximation of a computational theory of protocol correctness? Some initial work on this last problem has been done by Baudet et al. [25], but there is still much to be learned.

22.5 Systems and Algorithms

In this section, we give some representative examples of systems based on the algorithmic approaches just presented.

22.5.1 NPA and Maude-NPA

The NRL Protocol Analyzer, or NPA, was one of the earliest tools for verifying the security of cryptographic protocols. Although not originally designed as a model checker, it later took on many of the features of one, including the ability to check properties expressed in a temporal logic language, NPATRL [74]. In NPA, both the actions of honest agents and the adversary were specified in terms of state transitions. Model checking employed backward search, where the output of a state transition was unified with the current state. NPA had limited support for equational unification and could, for example, model properties such as the cancellation of encryption and decryption.

One of the most interesting properties of NPA was that it was an unbounded-session model checker. It included built-in inductive techniques for building grammars defining languages of infinite search paths. Once NPA reached a state that contained a term in the grammar, it would not explore beyond that state. This technique often allowed NPA to terminate after a finite number of steps, although of course termination was not guaranteed. It has been used to verify a number of protocols and protocol standards, including the Internet Key Exchange Protocol [72], the Group Domain of Interpretation Protocol [74], and the Simmons Selective Broadcast Protocol [70].

Maude-NPA is a descendant of NPA, implemented in the Maude rewriting language [36]. It shares many of NPA's features, including the use of unification to

implement backward search and reliance on grammars to ensure termination. There are two main differences. First, it is implemented in the Maude rewriting language, which gives it a formal basis in rewriting logic. In particular, backward search in Maude-NPA is implemented via narrowing over a simple transition model that is expressed via a small set of rewrite rules. Narrowing is a technique for deduction using rewrite rules and unification that, given a term t, finds a substitution σ that unifies a subterm with the left-hand side of a rewrite rule $l \rightarrow r$. The subterm $\sigma(l)$ of $\sigma(t)$ is replaced by $\sigma(r)$. When the rewrite rules describe state transitions, narrowing gives an algorithm for state-space exploration. When the arrows are reversed, as they are in Maude-NPA, narrowing can be used to implement backward search from a final goal. A second difference is that Maude-NPA is devoted to reasoning about the different equational theories that describe the behavior of cryptographic algorithms. Maude-NPA makes use of unification modulo different equational theories as the unification step in its narrowing algorithm.

Maude-NPA also incorporates two state-space reduction techniques that were originally used in NPA, but have been refined and extended in Maude-NPA. The first is subsumption-based partial-order reduction, in which the unreachability of a Maude-NPA state description is implied by the unreachability of another state description if a certain subsumption relation exists between them. The second is a super-lazy intruder. This is similar to the lazy intruder of the OFMC [20], except that it is adapted for backward search. If a variable term or a term constructed out of variable terms appears in the adversary knowledge part of the state, then the search proceeds no further on this term (that is, it is removed from the state), since the adversary should be able to find it using arbitrary terms.[2] However, it is not deleted entirely but kept around in a ghost state. If any of the variables in the term are instantiated, the ghost state is resuscitated. Among the equational theories that Maude-NPA can currently handle are a subclass of subterm-convergent theories, as well as exclusive-or, Abelian groups, modular exponentiation, bounded associativity, and homomorphic encryption over a free operator. Work is ongoing on incorporating more general homomorphic encryption.

22.5.2 AVISPA and Related Tools

The AVISPA tool [4] is a model checker that integrates several different model-checking approaches. AVISPA provides a high-level specification language, HLPSL, for specifying protocols and their properties. Protocols are specified in HLPSL in terms of their roles, using control flow patterns, data structures, and alternative adversary models, as well as different cryptographic primitives and their algebraic properties. HLPSL specifications have a declarative semantics based on Lamport's Temporal Logic of Actions [63] and an operational semantics defined in terms of

[2] Maude-NPA does not have secret types, so we assume that the adversary can create at least one term of any type.

a rewrite-based formalism called the intermediate format, or IF. Different model-checking backends interpret the IF and can be used for (bounded) verification or falsification.

The main backends in the AVISPA tool are the Constraint-Logic-based Attack Searcher CL-Atse, the On-the-Fly Model Checker OFMC, and the SAT-based Model Checker SAT-MC. We previously discussed SAT-MC in Sect. 22.4.3 and restrict our attention here to CL-Atse and OFMC.

CL-Atse, like the other AVISPA backends, operates on IF specifications of protocols. CL-Atse represents protocol states symbolically as collections of non-ground facts, which record the states of different threads, the messages sent to the network, and the adversary knowledge. In particular, constraints are used to describe what the different agents know and a constraint calculus is used to solve for what they can know, from messages previously exchanged, i.e., the calculus is used to solve a variant of the non-ground intruder deduction problem. CL-Atse was designed to allow the easy integration of new deduction rules and operator properties. In particular, CL-Atse integrates a version of Baader and Schulz's unification algorithm [8] with modules for xor, exponentiation, and associative pairing.

OFMC combines a number of techniques to enable the efficient analysis of security protocols. First, OFMC uses lazy data types (in a functional programming setting) as a simple way of building efficient on-the-fly model checkers for protocols with very large, or even infinite, state spaces. A lazy data type is one where data constructors (such as *cons* for building lists or *node* for building trees) build data without evaluating their arguments; this allows one to represent and compute with infinite data (e.g., streams or infinite trees), generating arbitrary prefixes of the data on demand. In [14], lazy data types are used to build, and compute with, models of security protocols: a protocol and a description of the powers of an adversary are formalized as an infinite tree. Lazy evaluation is used to decouple the model from search and heuristics, building the infinite tree on the fly, in a demand-driven fashion. Second, OFMC models the adversary in a lazy fashion (the so-called "lazy intruder"), where adversary communication is represented symbolically and solved during search. To this end, like CL-Atse, it integrates a constraint solver for the non-ground intruder deduction problem. Effectively, OFMC performs search at two levels: search in the space of symbolic states, and search in the space of constraints. Third, while OFMC performs verification for a bounded number of sessions, it works with *symbolic session generation* which avoids enumerating all possible ways of instantiating possible sessions. Fourth, OFMC exploits a state-space reduction technique, inspired by partial-order reduction, called constraint differentiation [81]. Constraint differentiation works by eliminating certain kinds of redundancies that arise in the search space when using constraints to represent and manipulate the messages that may be sent by the adversary. Namely, different symbolic states may describe overlapping sets of ground states. Constraint differentiation essentially computes a set-difference symbolically, to minimize these overlaps. This can be seen as generalizing the kind of subsumption-based partial-order reduction used in Maude-NPA. Finally, OFMC also provides some limited support for handling different equationally specified operators on messages [19].

22.5.3 Athena, Scyther, and Tamarin

The Athena [94] and Scyther [42] algorithms implement model checking with respect to the unbounded model described in Sect. 22.3.5 by performing a backward-style search. For these methods, the model is extended with *adversary events* for *encrypting*, *decrypting*, *hashing*, and *knowing* messages. Infinite sets of states are represented by *(trace) patterns*: partially ordered sets of events that must occur in the traces, and whose messages may contain variables. (In the Athena model, patterns are referred to as *semi-bundles*.) The events in patterns must satisfy a number of criteria that follow from the semantics. For example, if an event occurs in the pattern from a role R with thread identifier tid, then (a) the pattern does not contain events from other roles with thread identifier tid and (b) the event is preceded in the pattern by all events that precede it in the role R, with identical substitutions and thread identifier tid. However, it is not required for receive events in patterns that the received term can be inferred from the union of the initial knowledge and the messages that occur in preceding send events within the pattern. Patterns allow for specifying properties such as secrecy.

Example 4 (Secrecy pattern) The following pattern PT specifies the violation of secrecy of the nonce of the responder role of the Needham–Schroeder protocol, performed by b when trying to communicate with a, where X and tid are variables. The AdversaryKnows event is used to encode the secrecy violation.

$$
\begin{aligned}
e_1 &= \big(tid, \mathsf{recv}\big(a, b, \{\!|\, a, X \,|\!\}^{\mathsf{a}}_{pk(b)}\big)\big)\\
e_2 &= \big(tid, \mathsf{send}\big(b, a, \{\!|\, X, N_B \sharp tid \,|\!\}^{\mathsf{a}}_{pk(a)}\big)\big)\\
e_3 &= \big(tid, \mathsf{recv}\big(a, b, \{\!|\, N_B \sharp tid \,|\!\}^{\mathsf{a}}_{pk(b)}\big)\big)\\
e_4 &= \mathsf{AdversaryKnows}(N_B \sharp tid)\\
PT &= \big(\{e_1, e_2, e_3, e_4\}, \big\{(e_1, e_2), (e_2, e_3)\big\}^+\big)
\end{aligned}
$$

This pattern represents an infinite set of traces of the Needham–Schroeder protocol, e.g., the attack from Fig. 2 and all traces that additionally include arbitrarily interleaved threads. In contrast, the corresponding pattern for the initiator role (also with agents a and b) represents the empty set of traces.

By introducing restrictions on variable instantiations into the algorithm, and replacing a and b by variables that can only be instantiated by non-compromised agents, we can faithfully represent all violations of the secrecy property for the protocol in the above example.

During backward search, a case distinction on the source of messages is used for branching and the patterns are extended by either adding events, adding ordering constraints, or unifying terms. The search can terminate in two ways. First, the pattern can be proven to be empty, i.e., it contains no traces of the protocol. The

main mechanism here is detecting cyclic dependencies of the messages. Second, the receive events in the pattern meet all premises of the receive event: the adversary can produce an appropriate message from the preceding events. In such a case, the pattern is called *realizable* and it corresponds to an infinite set of actual traces; a representative trace (of minimal length) from this set can be generated by linearizing the non-adversary events and instantiating the remaining variables from the adversary knowledge.

Scyther differs from Athena in how it makes the case distinction and in the possible outcomes of the analysis. With respect to the possible outcomes, Scyther bounds the size of the patterns but detects whether the bound is reached. By bounding the size of the patterns, termination is guaranteed and one of three possible results occurs. First, if a realizable pattern is found, a representative (attack) trace is constructed. Second, if no realizable patterns are found, and the bound is not reached, no realizable patterns exist (for any bound). In case of an attack pattern, this corresponds to the absence of attacks. Third, if no realizable patterns are found but the bound is reached, the result can be interpreted as verification with respect to a bounded number of sessions and is similar to the guarantees provided by, e.g., OFMC or CL-Atse when they do not find attacks.

The Tamarin prover [90] is a generalization of the algorithms underlying Athena and Scyther. Tamarin uses a backward search that can handle more expressive protocol and property specifications. Protocols and adversary capabilities can be specified using multiset rewriting rules, allowing the specification of protocols with branching and loops. Tamarin provides support for Diffie–Hellman exponentiations and a class of user-defined equational theories. Protocols can be analyzed with respect to properties specified in a guarded fragment of first-order logic that supports quantification over timepoints.

22.5.4 ProVerif

The ProVerif tool [29] uses abstractions to obtain an efficient analysis method. In particular, it employs two main abstractions compared to the operational semantics presented before. First, individual fresh values are abstracted into sets of fresh values. Second, each action of a thread can be executed multiple times.

The abstracted protocol model can be represented as a set of Horn clauses. These Horn clauses are analyzed using a two-phase resolution algorithm. In the first phase, the Horn clauses are saturated in a forward fashion until a fixpoint is reached. This phase combines multiple derivations into single rules, thereby optimizing the rule set. Facts can be derived from the optimized rule set if and only if they can be derived from the original rule set. In the second phase, a backward depth-first search is used to try to establish that a fact (usually representing the adversary knowing a message that is supposed to be secret) cannot be derived from the saturated Horn clauses. If this cannot be established because the fact can be derived, an attempt is made to reconstruct a corresponding protocol trace.

If we assume that the fact represents a secret, we can interpret each of the four possible results of running the tool in the following way. First, if the fact cannot be derived, the overapproximation ensures that no protocol execution will leak the secret, and hence the protocol satisfies secrecy. Second, if the fact can be derived and the derivation can be translated into a corresponding (attack) trace, the protocol does not satisfy secrecy, as witnessed by the trace. Third, if the fact can be derived but no corresponding trace can be reconstructed from the derivation, the result is inconclusive. Finally, either of the two phases in the algorithm may not terminate and again no knowledge is gained about the security of the protocol.

ProVerif can handle authentication properties formalized as correspondence properties [30]. These properties express that when event e_1 occurs, event e_2 must have occurred earlier with related parameters ρ_1 and ρ_2. For example ρ_1 could be a nonce and ρ_2 a variable that is supposed to be instantiated with ρ_1. In this case, it does not suffice to prove that the values of ρ_1 and ρ_2 abstract into the same set (i.e., they are in the same equivalence class in the abstraction) and a finer abstraction is used. By introducing session identifiers in the construction of ρ_1 and ρ_2, we increase the precision of the verification at the cost of efficiency and termination.

22.6 Research Problems

Although the research community has come far in recent decades, many research challenges remain. Below we describe some of the most pressing problems being tackled as well as open problems.

22.6.1 Link to Computational Soundness

The Dolev–Yao model, even when equational properties are added, treats cryptosystems as black boxes. If the adversary possesses the appropriate key, he can learn the contents of an encrypted term. Otherwise he is completely ignorant of the corresponding plaintext. Moreover he is only able to perform the operations specified in the protocol. This is very different from definitions used by cryptographers. In these definitions, the adversary is modeled as a probabilistic polynomial time Turing machine. The security properties of the cryptosystems themselves vary depending on what kinds of attacks they are assumed to be secure against, e.g., chosen plaintext or chosen ciphertext. Finally, secrecy is usually specified not in terms of the adversary not being able to obtain a given secret, but in the indistinguishability between two different versions of the protocol, e.g., two versions with different secrets, or one constructed using a secret and one constructed using random data, or in the indistinguishability between the real secret and a random bit string. Is it possible to come up with an approach to proving protocols correct that combines the amenability to exhaustive search of the Dolev–Yao model with the stronger requirements of cryptographic models?

There has been a substantial amount of work on this problem. The general idea is to have two models, a symbolic model and a computational model, and establish some sort of relationship between them, e.g., a simulation relation, so that security in the symbolic model implies security in the computational model. In some approaches, the symbolic specification is trivially secure, while in others it is a Dolev–Yao-style specification that can be verified with a model checker. A survey of research in this area is given by Cortier et al. in [41].

Although research in this area has been successful in establishing links between the two models, the results have been criticized both for being so complex as to detract from the advantage of being able to reason at the simpler Dolev–Yao level, and for being too limited in the types of cryptosystems they can handle. For example, the reactive simulatability framework of Backes et al. [11], one of the most prominent models in this area, has been shown to be impossible to extend to include two standard items in the cryptographer's toolbox: one-way hash functions [10] and exclusive-or [9]. This is perhaps not surprising, given the divergence between the two models and the difficulty of bringing the two together. Hence this is still an active area of research.

22.6.2 Corruption Models

In the basic Dolev–Yao model described in Sect. 22.3, the Dolev–Yao adversary initially has the long-term keys of some of the agents. This formalizes a notion of static corruption where the adversary has compromised some of the agents and can play as an "insider" during protocol execution. However, many protocols are intended to be secure against much more sophisticated corruption models. [33, 58, 93], for example, specify adversaries who can dynamically corrupt long-term secrets, session keys and other parts of the session state, or even random number generators. For example, a Diffie–Hellman key agreement protocol, where digital signatures are used to authenticate the exchanged half-keys, provides perfect forward secrecy [75]: the resulting key remains secret even when the signature keys are later compromised by the adversary.

Some of the earlier model checkers, such as the NRL Protocol Analyzer, allowed for the dynamic corruption of keys. More recently in [15, 16], Basin and Cremers have examined this issue more thoroughly by formalizing a hierarchy of corruption models. Their models extend the operational semantics presented earlier and cover many aspects of the adversary models used in cryptographic models. The Scyther tool supports the evaluation of protocols with respect to these corruption models, which has led to the automatic discovery of many attacks that previously could only be found by manual analysis.

The additional complexity of richer corruption models significantly increases the time required for verification. To make analysis of larger protocols with respect to these models feasible, it would be useful to develop more efficient dedicated model-checking algorithms. Furthermore, traditional compositionality results, e.g., [3, 53],

no longer hold under stronger corruption models. Hence, another research challenge is to develop analogous compositionality results for this purpose.

An alternative direction is to weaken the adversary models. In fact, in many cases, protocols are designed under the assumption that adversaries are not completely dishonest and for various reasons do not perform all the activities available to the Dolev–Yao adversary. One example is the *honest but curious* agent, who acts according to the rules of the protocol, but attempts to learn secret information from the messages that it has received legitimately. A related case is the *passive adversary* who does not even participate in the protocol but simply observes passing traffic. Other protocols, including for example many electronic commerce protocols, are based on the assumption that an adversary will not take any action against its own best interests, such as those that involve revealing its own secret information. Although a fair amount of work has been done on model checking protocols with respect to these various adversary models, a more comprehensive approach, in which the user could specify which adversary model will be used, would be of benefit.

22.6.3 Channel Properties

The standard Dolev–Yao model gives the adversary control over all communication channels. The adversary sees all communications, and can block, alter, and redirect traffic at will. Thus all an agent can conclude from receipt of information sent along one of these channels is that somebody sent it. Assurance of other properties must be gained by cryptographic means. However, a growing number of cryptographic protocols either use channels with special properties or rely upon assumptions about weaker adversaries. These include anonymous routing protocols such as Tor, which assume an adversary who is able to spy on only part of the network, distance bounding protocols [32] and secure localization protocols [99], which rely on the use of timed wireless communications to verify that a prover is within a certain range of a verifier, and protocols that rely on human-verifiable channels [13], such as a human reading a sequence of numbers off a computer screen, to bootstrap key distribution in the absence of a public key infrastructure. Work has been ongoing on developing methods for formal analysis of protocols that use these channels, e.g., [21, 73, 89, 97], but most of it has not yet been applied to model checking, concentrating more on theorem proving or specialized logics. The problem of how best to model and reason about these channels using a model checker is still open.

22.6.4 Other Properties, Including Non-trace Properties

The application of formal methods to cryptographic protocol analysis was originally restricted to the study of various forms of secrecy and authentication. These are straightforward to formulate using temporal logics and analyze using model

checkers. However, there are other classes of properties that are less straightforward. Often these are properties that are not trace properties themselves, but can be approximated by trace properties such that if the trace property holds, then so does the original property.

One of the earliest properties of this type is *non-interference*, which, roughly speaking, is the property that events labeled "high" should have no discernible effect on events labeled "low". Non-interference is closely related to the cryptographic notion of indistinguishability, mentioned in Sect. 22.6.1. Indistinguishability formalizes that an adversary should be unable to distinguish between two protocols, usually either the same protocol using different secrets, or a real protocol and a simulation of the protocol using random data. This can be approximated by a trace-based notion called *observational equivalence*, which has been implemented in ProVerif by running the two protocols in tandem and checking for equivalence at each transition [31].

Another property related to indistinguishability is *static equivalence* [2]. This notion is defined with respect to an underlying equational theory and, roughly speaking, two terms are statically equivalent when they satisfy the same equations. As noted in [26], static equivalence is essentially a special case of observational equivalence that does not allow for continued interaction between a system and an observer: the observer gets data once and conducts experiments on its own. Static equivalence has direct application to modeling off-line guessing attacks, which are attacks where the adversary tries to guess a secret and verify his guess, without further communication. As shown in [22, 40], static equivalence may be used to specify the absence of off-line guessing attacks by expressing that the adversary cannot distinguish between two versions of the same symbolic trace: one corresponding to a correct guess and the other corresponding to an incorrect guess. Decision procedures for static equivalence have been implemented by the YAPA [24], KISS [34], and FAST [39] tools. ProVerif also supports the analysis of off-line guessing attacks, but based on a different formalization of guessing due to [40].

We see that there have been a number of individual solutions to reasoning about special properties. But what would be useful to have is a more general approach to such properties that could be tailored to specific instances. For example, as we have seen from the above, many security properties are expressed in terms of some sort of equivalence between families of traces. One might expect a general procedure to exist for such equivalences.

22.6.5 Probabilities

As noted previously, probabilities are part of standard cryptographic definitions of security. Probabilities also arise when security protocols are based on randomized algorithms or the protocol guarantees themselves are probabilistic. Different options for augmenting transition systems and logics with probabilities are discussed

in Chap. 28 of this handbook [12]. Probabilistic model checking has been successfully applied to different kinds of security protocols. Examples include protocols for anonymity [92], non-repudiation [64], and contract signing [84].

As an example, in [92], Shmatikov models the protocol underlying Crowds [87], which is a peer-to-peer group communication system based on random message routing among members. He models the behavior of group members and the adversary as a discrete-time Markov chain, and formalizes the required anonymity properties of the system in the probabilistic temporal logic PCTL. Given this model, he can use the PRISM model checker [55] to analyze the anonymity provided by the system, showing, for example, how probabilistic anonymity degrades as the group size increases.

Until now, work on model checking stochastic systems and model checking security has been disjoint. The problem of using off-the-shelf general model checkers, whether for stochastic systems or other classes of systems, is that they neither offer optimizations useful in the domain of security (e.g., for handling the Dolev–Yao intruder) nor support cryptographic operators and equational extensions. An open problem is how best to combine the models and algorithms from these two areas.

22.7 Conclusions

We have shown in this chapter how to extend transition-system models of concurrent computation to model cryptographic protocols. The main extensions have been with a term language formalizing cryptographic messages and a model of a network (Dolev–Yao) adversary. These extensions themselves are fairly straightforward but they lead to two immediate challenges: the resulting system has infinitely many states and the active adversary results in considerable nondeterminism. Addressing these challenges has motivated a number of specialized model-checking techniques.

State of the art methods and tools are able to handle classical authentication and key-exchange protocols of realistic, but limited, complexity. Abstract models of large protocol suites such as the Internet Key Exchange protocol have been analyzed. However, an automatic analysis of the full protocol suite, with all its variations, Diffie–Hellman equational reasoning, and a full model of its branching and looping behaviors, would lead to a state-space explosion and is beyond the state of the art. A state-space explosion arises due to a combination of the protocol's size and the cryptographic operators used. In the short term, improved support for equational reasoning will make a difference; in the mid-term and long term, better support for reasoning using abstraction is required.

Security protocols go far beyond authentication and key exchange in practice. We have sketched some of the challenges in handling more precise, cryptographic notions of security as well as wider classes of cryptographic primitives and properties. Interestingly, advances in other areas of model checking, such as probabilistic model checking, may play an important role in enabling new classes of applications.

References

1. Abadi, M., Cortier, V.: Deciding knowledge in security protocols under equational theories. Theor. Comput. Sci. **367**(1–2), 2–32 (2006)
2. Abadi, M., Fournet, C.: Mobile values, new names, and secure communication. In: Hankin, C., Schmidt, D. (eds.) Symp. on Principles of Programming Languages (POPL), pp. 104–115. ACM, New York (2001)
3. Andova, S., Cremers, C., Gjøsteen, K., Mauw, S., Mjølsnes, S.F., Radomirović, S.: A framework for compositional verification of security protocols. Inf. Comput. **206**, 425–459 (2008)
4. Armando, A., Basin, D., Boichut, Y., Chevalier, Y., Compagna, L., Cuéllar, J., Drielsma, P.H., Héam, P.C., Kouchnarenko, O., Mantovani, J., Mödersheim, S., von Oheimb, D., Rusinowitch, M., Santiago, J., Turuani, M., Viganò, L., Vigneron, L.: The AVISPA tool for the automated validation of internet security protocols and applications. In: Etessami, K., Rajamani, S.K. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 3576, pp. 281–285. Springer, Heidelberg (2005)
5. Armando, A., Compagna, L.: SATMC: a SAT-based model checker for security protocols. In: Alferes, J.J., Leite, J.A. (eds.) Logics in Artificial Intelligence—Journées Européennes sur la Logique en Intelligence Artificielle (JELIA). LNCS, vol. 3229, pp. 730–733. Springer, Heidelberg (2004)
6. Armando, A., Compagna, L.: SAT-based model-checking for security protocols analysis. Int. J. Inf. Secur. **7**(1), 3–32 (2008)
7. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press, Cambridge (1998)
8. Baader, F., Schulz, K.U.: Unification in the union of disjoint equational theories: combining decision procedures. J. Symb. Comput. **21**(2), 211–243 (1996)
9. Backes, M., Pfitzmann, B.: Limits of the cryptographic realization of Dolev–Yao-style XOR. In: di Vimercati, S.D.C., Syverson, P.F., Gollmann, D. (eds.) European Conf. on Research in Computer Security (ESORICS). LNCS, vol. 3679, pp. 178–196. Springer, Heidelberg (2005)
10. Backes, M., Pfitzmann, B., Waidner, M.: Limits of the BRSIM/UC soundness of Dolev–Yao models with hashes. In: Gollmann, D., Meier, J., Sabelfeld, A. (eds.) European Conf. on Research in Computer Security (ESORICS). LNCS, vol. 4189, pp. 404–423. Springer, Heidelberg (2006)
11. Backes, M., Pfitzmann, B., Waidner, M.: The reactive simulatability (RSIM) framework for asynchronous systems. Inf. Comput. **205**(12), 1685–1720 (2007)
12. Baier, C., de Alfaro, L., Forejt, V., Kwiatkowska, M.: Model checking probabilistic systems. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
13. Balfanz, D., Smetters, D.K., Stewart, P., Wong, H.C.: Talking to strangers: authentication in ad-hoc wireless networks. In: Network and Distributed System Security Symp. (NDSS) (2002)
14. Basin, D.: Lazy infinite-state analysis of security protocols. In: Secure Networking—CQRE [Secure] '99. LNCS, vol. 1740, pp. 30–42. Springer, Heidelberg (1999)
15. Basin, D., Cremers, C.: Degrees of security: protocol guarantees in the face of compromising adversaries. In: Dawar, A., Veith, H. (eds.) Intl. Workshop Computer Science Logic (CSL). LNCS, vol. 6247, pp. 1–18. Springer, Heidelberg (2010)
16. Basin, D., Cremers, C.: Modeling and analyzing security in the presence of compromising adversaries. In: Gritzalis, D., Preneel, B., Theoharidou, M. (eds.) European Conf. on Research in Computer Security (ESORICS). LNCS, vol. 6345, pp. 340–356. Springer, Heidelberg (2010)
17. Basin, D., Cremers, C., Meier, S.: Provably repairing the ISO/IEC 9798 standard for entity authentication. In: Degano, P., Guttman, J.D. (eds.) Intl. Conf. Principles of Security and Trust (POST). LNCS, vol. 7215, pp. 129–148. Springer, Heidelberg (2012)
18. Basin, D., Ganzinger, H.: Automated complexity analysis based on ordered resolution. J. Assoc. Comput. Mach. **48**(1), 70–109 (2001)

19. Basin, D., Mödersheim, S., Viganò, L.: Algebraic intruder deductions. In: Sutcliffe, G., Voronkov, A. (eds.) Intl. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR). Lecture Notes in Artificial Intelligence, vol. 3835, pp. 549–564. Springer, Heidelberg (2005)
20. Basin, D., Mödersheim, S., Viganò, L.: OFMC: a symbolic model checker for security protocols. Int. J. Inf. Secur. **4**(3), 181–208 (2005)
21. Basin, D., Čapkun, S., Schaller, P., Schmidt, B.: Let's get physical: models and methods for real-world security protocols. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) Intl. Conf. on Theorem Proving in Higher Order Logics (TPHOLs). LNCS, vol. 5674, pp. 1–22. Springer, Heidelberg (2009). Invited paper
22. Baudet, M.: Deciding security of protocols against off-line guessing attacks. In: Atluri, V., Meadows, C., Juels, A. (eds.) ACM Conf. on Computer and Communications Security (CCS), pp. 16–25. ACM, New York (2005)
23. Baudet, M., Cortier, V., Delaune, S.: YAPA: a generic tool for computing intruder knowledge. In: Treinen, R. (ed.) Intl. Conf. Rewriting Techniques and Applications (RTA). LNCS, vol. 5595, pp. 148–163. Springer, Heidelberg (2009)
24. Baudet, M., Cortier, V., Delaune, S.: YAPA: a generic tool for computing intruder knowledge. In: Treinen, R. (ed.) Intl. Conf. Rewriting Techniques and Applications (RTA), pp. 148–163. Springer, Heidelberg (2009)
25. Baudet, M., Cortier, V., Kremer, S.: Computationally sound implementations of equational theories against passive adversaries. Inf. Comput. **207**(4), 496–520 (2009)
26. Baudet, M., Warinschi, B., Abadi, M.: Guessing attacks and the computational soundness of static equivalence. J. Comput. Secur. **18**(5), 909–968 (2010)
27. Bhargavan, K., Fournet, C., Corin, R., Zalinescu, E.: Cryptographically verified implementations for TLS. In: Ning, P., Syverson, P.F., Jha, S. (eds.) ACM Conf. on Computer and Communications Security (CCS), pp. 459–468. ACM, New York (2008)
28. Bhargavan, K., Fournet, C., Gordon, A.D., Swamy, N.: Verified implementations of the information card federated identity-management protocol. In: Abe, M., Gligor, V.D. (eds.) Symp. on Information, Computer and Communications Security (ASIACCS), pp. 123–135. ACM, New York (2008)
29. Blanchet, B.: An efficient cryptographic protocol verifier based on Prolog rules. In: Computer Security Foundations Workshop (CSFW), pp. 82–96. IEEE, Piscataway (2001)
30. Blanchet, B.: Automatic verification of correspondences for security protocols. J. Comput. Secur. **17**(4), 363–434 (2009)
31. Blanchet, B., Abadi, M., Fournet, C.: Automated verification of selected equivalences for security protocols. In: Symp. on Logic in Computer Science (LICS), pp. 331–340. IEEE, Piscataway (2005)
32. Brands, S., Chaum, D.: Distance-bounding protocols. In: Helleseth, T. (ed.) Workshop on the Theory and Application of Cryptographic Techniques (EUROCRYPT '93), pp. 344–359. Springer, Heidelberg (1994)
33. Canetti, R., Krawczyk, H.: Analysis of key-exchange protocols and their use for building secure channels. In: Pfitzmann, B. (ed.) Intl. Conf. on the Theory and Application of Cryptographic Techniques (EUROCRYPT). LNCS, vol. 2045, pp. 453–474. Springer, Heidelberg (2001)
34. Ciobâca, S., Delaune, S., Kremer, S.: Computing knowledge in security protocols under convergent equational theories. In: Schmidt, R.A. (ed.) Intl. Conf. on Automated Deduction (CADE), pp. 355–370. Springer, Heidelberg (2009)
35. Clarke, E.M., Jha, S., Marrero, W.R.: Verifying security protocols with Brutus. Trans. Softw. Eng. Methodol. **9**(4), 443–487 (2000)
36. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L.: All About Maude—A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic. LNCS, vol. 4350. Springer, Heidelberg (2007)

37. Comon-Lundh, H., Delaune, S.: The finite variant property: how to get rid of some algebraic properties. In: Giesl, J. (ed.) Intl. Conf. Rewriting Techniques and Applications (RTA). LNCS, vol. 3467, pp. 294–307. Springer, Heidelberg (2005)
38. Comon-Lundh, H., Treinen, R.: Easy intruder deductions. In: Dershowitz, N. (ed.) Verification: Theory and Practice, Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday. LNCS, vol. 2772, pp. 225–242. Springer, Heidelberg (2003)
39. Conchinha, B., Basin, D., Caleiro, C.: Efficient decision procedures for message deducibility and static equivalence. In: Degano, P., Etalle, S., Guttman, J.D. (eds.) Intl. Workshop on Formal Aspects of Security and Trust (FAST 2010). LNCS, vol. 6561, pp. 34–49 (2011)
40. Corin, R., Doumen, J., Etalle, S.: Analysing password protocol security against off-line dictionary attacks. Electron. Notes Theor. Comput. Sci. **121**, 47–63 (2005)
41. Cortier, V., Kremer, S., Warinschi, B.: A survey of symbolic methods in computational analysis of cryptographic systems. J. Autom. Reason. **46**(3–4), 225–259 (2011)
42. Cremers, C.: The Scyther tool: verification, falsification, and analysis of security protocols. In: Gupta, A., Malik, S. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 5123, pp. 414–418. Springer, Heidelberg (2008)
43. Cremers, C.: Key exchange in IPsec revisited: formal analysis of IKEv1 and IKEv2. In: Atluri, V., Díaz, C. (eds.) European Conf. on Research in Computer Security (ESORICS), pp. 315–334. Springer, Heidelberg (2011)
44. Dierks, T., Rescorla, E.: The transport layer security (TLS) protocol version 1.1. Tech. Rep. RFC 4346, Internet Engineering Task Force (2006)
45. Dolev, D., Even, S., Karp, R.M.: On the security of ping-pong protocols. Inf. Control **55**(1–3), 57–68 (1982)
46. Dolev, D., Yao, A.C.C.: On the security of public key protocols (extended abstract). In: Annual Symp. on Foundations of Computer Science (FOCS), pp. 350–357. IEEE, Piscataway (1981)
47. Dolev, D., Yao, A.C.C.: On the security of public key protocols. Trans. Inf. Theory **29**(2), 198–207 (1983)
48. Durgin, N.A., Lincoln, P., Mitchell, J.C.: Multiset rewriting and the complexity of bounded security protocols. J. Comput. Secur. **12**(2), 247–311 (2004)
49. Durgin, N.A., Lincoln, P., Mitchell, J.C., Scedrov, A.: Undecidability of bounded security protocols. In: Workshop on Formal Methods and Security Protocols (1999)
50. Escobar, S., Meadows, C., Meseguer, J.: Maude-NPA: cryptographic protocol analysis modulo equational properties. In: Aldini, A., Barthe, G., Gorrieri, R. (eds.) Foundations of Security Analysis and Design (FOSAD), 2007/2008/2009 Tutorial Lectures. LNCS, vol. 5705, pp. 1–50. Springer, Heidelberg (2007)
51. Escobar, S., Meseguer, J., Sasse, R.: Variant narrowing and equational unification. Electron. Notes Theor. Comput. Sci. **238**(3), 103–119 (2009)
52. Even, S., Goldreich, O., Shamir, A.: On the security of ping-pong protocols when implemented using the RSA. In: Williams, H.C. (ed.) Intl. Cryptology Conf. (CRYPTO). LNCS, vol. 218, pp. 58–72. Springer, Heidelberg (1985)
53. Guttman, J., Thayer, F.: Protocol independence through disjoint encryption. In: Computer Security Foundations Workshop (CSFW), pp. 24–34. IEEE, Piscataway (2000)
54. Harkins, D., Carrel, D., et al.: The internet key exchange (IKE) (1998)
55. Hinton, A., Kwiatkowska, M., Norman, G., Parker, D.: PRISM: a tool for automatic verification of probabilistic systems. In: Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), vol. 3920, pp. 441–444 (2006)
56. Holzmann, G.: The model checker SPIN. Trans. Softw. Eng. **23**(5), 279–295 (2002)
57. Hopcroft, P.J., Lowe, G.: Analysing a stream authentication protocol using model checking. Int. J. Inf. Secur. **3**(1), 2–13 (2004)
58. Just, M., Vaudenay, S.: Authenticated multi-party key agreement. In: Kim, K., Matsumoto, T. (eds.) Advances in Cryptology (ASIACRYPT '96). LNCS, vol. 1163, pp. 36–49 (1996)
59. Kaufman, C., Hoffman, P., Nir, Y., Eronen, P.: Internet key exchange protocol version 2 (IKEV2). Tech. Rep. RFC 5996, Internet Engineering Task Force (September 2010)

60. Kremer, S., Ryan, M., Smyth, B.: Election verifiability in electronic voting protocols. In: Gritzalis, D., Preneel, B., Theoharidou, M. (eds.) European Conf. on Research in Computer Security (ESORICS). LNCS, vol. 6345, pp. 389–404. Springer, Heidelberg (2010)
61. Küsters, R., Truderung, T.: Reducing protocol analysis with XOR to the XOR-free case in the Horn theory based approach. In: Ning, P., Syverson, P.F., Jha, S. (eds.) ACM Conf. on Computer and Communications Security (CCS), pp. 129–138. ACM, New York (2008)
62. Küsters, R., Truderung, T.: Using ProVerif to analyze protocols with Diffie-Hellman exponentiation. In: Computer Security Foundations Symp. (CSF), pp. 157–171. IEEE, Piscataway (2009)
63. Lamport, L.: The temporal logic of actions. Trans. Program. Lang. Syst. **16**(3), 872–923 (1994)
64. Lanotte, R., Maggiolo-Schettini, A., Troina, A.: Automatic analysis of a non-repudiation protocol. Electron. Notes Theor. Comput. Sci. **112**, 113–129 (2005)
65. Longley, D., Rigby, S.: An automatic search for security flaws in key management schemes. Comput. Secur. **11**(1), 75–89 (1992)
66. Lowe, G.: Breaking and fixing the Needham–Schroeder public-key protocol using FDR. Softw., Concepts Tools **17**(3), 93–102 (1996)
67. Lowe, G.: A hierarchy of authentication specifications. In: Computer Security Foundations Workshop (CSFW), pp. 31–44 (1997)
68. Lowe, G.: Casper: a compiler for the analysis of security protocols. J. Comput. Secur. **6**(1–2), 53–84 (1998)
69. Marques-Silva, J., Malik, S.: Propositional SAT solving. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
70. Meadows, C.: Applying formal methods to the analysis of a key management protocol. J. Comput. Secur. **1**(1), 5–36 (1992)
71. Meadows, C.: The NRL Protocol Analyzer: an overview. J. Log. Program. **26**(2), 113–131 (1996)
72. Meadows, C.: Analysis of the Internet Key Exchange protocol using the NRL Protocol Analyzer. In: Symp. on Security and Privacy (S & P), pp. 216–231. (1999)
73. Meadows, C., Poovendran, R., Pavlovic, D., Chang, L., Syverson, P.: Distance bounding protocols: authentication logic analysis and collusion attacks. In: Poovendran, R., Wang, C., Roy, S. (eds.) Secure Localization and Time Synchronization in Wireless Ad Hoc and Sensor Networks. Springer, Heidelberg (2006)
74. Meadows, C., Syverson, P.F., Cervesato, I.: Formal specification and analysis of the Group Domain of Interpretation Protocol using NPATRL and the NRL Protocol Analyzer. J. Comput. Secur. **12**(6), 893–931 (2004)
75. Menezes, A., van Oorschot, P., Vanstone, S.: Handbook of Applied Cryptography. CRC Press, Boca Raton (1996)
76. Millen, J.K.: A necessarily parallel attack. In: Workshop on Formal Methods and Security Protocols (1999)
77. Millen, J.K.: On the freedom of decryption. Inf. Process. Lett. **86**(6), 329–333 (2003)
78. Millen, J.K., Clark, S.C., Freedman, S.B.: The Interrogator: protocol security analysis. Trans. Softw. Eng. **13**(2), 274–288 (1987)
79. Millen, J.K., Shmatikov, V.: Constraint solving for bounded-process cryptographic protocol analysis. In: Reiter, M.K., Samarati, P. (eds.) ACM Conf. on Computer and Communications Security (CCS), pp. 166–175 (2001)
80. Mitchell, J.C., Mitchell, M., Stern, U.: Automated analysis of cryptographic protocols using Murphi. In: Symp. on Security and Privacy (S & P), pp. 141–151. IEEE, Piscataway (1997)
81. Mödersheim, S., Viganò, L., Basin, D.: Constraint differentiation: search-space reduction for the constraint-based analysis of security protocols. J. Comput. Secur. **18**(4), 575–618 (2010)
82. Needham, R.M., Schroeder, M.D.: Using encryption for authentication in large networks of computers. Commun. ACM **21**(12), 993–999 (1978)
83. Neuman, C., Hartman, S., Raeburn, K.: The Kerberos network authentication service (V5). Tech. Rep. RFC 4120, Internet Engineering Task Force (July 2005)

84. Norman, G., Shmatikov, V.: Analysis of probabilistic contract signing. J. Comput. Secur. **14**(6), 561–589 (2006)
85. Peled, D.: Partial-order reduction. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
86. Perrig, A., Tygar, J.D.: Secure Broadcast Communication in Wired and Wireless Networks. Kluwer Academic, Norwell (2002)
87. Reiter, M., Rubin, A.: Crowds: anonymity for web transactions. Trans. Inf. Syst. Secur. **1**(1), 66–92 (1998)
88. Rusinowitch, M., Turuani, M.: Protocol insecurity with finite number of sessions is NP-complete. In: Computer Security Foundations Workshop (CSFW), p. 174. IEEE, Piscataway (2001)
89. Schaller, P., Schmidt, B., Basin, D., Čapkun, S.: Modeling and verifying physical properties of security protocols for wireless networks. In: Computer Security Foundations Symp. (CSF), pp. 109–123. IEEE, Piscataway (2009)
90. Schmidt, B., Meier, S., Cremers, C., Basin, D.: Automated analysis of Diffie-Hellman protocols and advanced security properties. In: Computer Security Foundations Symp. (CSF), pp. 78–94 (2012)
91. Schmidt-Schauß, M.: Unification in permutative theories is undecidable. J. Symb. Comput. **8**, 415–421 (1989)
92. Shmatikov, V.: Probabilistic analysis of an anonymity system. J. Comput. Secur. **12**(3), 355–377 (2004)
93. Shoup, V.: On formal models for secure key exchange (version 4) (1999). Revision of IBM Research Report RZ 3120, (April 1999)
94. Song, D.X.A.: A new efficient automatic checker for security protocol analysis. In: Computer Security Foundations Workshop (CSFW), pp. 192–202 (1999)
95. Stern, U., Dill, D.L.: Improved probabilistic verification by hash compaction. In: Camurati, P.E., Eveking, H. (eds.) Correct Hardware Design and Verification Methods. LNCS, vol. 987, pp. 206–224. Springer, Heidelberg (1995)
96. Syverson, P.F., Meadows, C.: A formal language for cryptographic protocol requirements. Des. Codes Cryptogr. **7**(1–2), 27–59 (1996)
97. Thayer, F.J., Swarup, V., Guttman, J.D.: Metric strand spaces for locale authentication protocols. In: Nishigaki, M., Jøsang, A., Murayama, Y., Marsh, S. (eds.) IFIP International Conference on Trust Management (IFIPTM), vol. 321, pp. 79–94. Springer, Heidelberg (2010)
98. Turuani, M.: The CL-Atse protocol analyser. In: Pfenning, F. (ed.) Intl. Conf. Rewriting Techniques and Applications (RTA). LNCS, vol. 4098, pp. 277–286. Springer, Heidelberg (2006)
99. Čapkun, S., Hubaux, J.: Secure positioning in wireless networks. J. Sel. Areas Commun. **24**(2), 221–232 (2006)
100. Viganò, L.: Automated security protocol analysis with the AVISPA tool. Electron. Notes Theor. Comput. Sci. **155**, 61–86 (2006)

Chapter 23
Transfer of Model Checking to Industrial Practice

Robert P. Kurshan

Abstract This chapter traces the practical challenges that were overcome in order to transfer model-checking theory to industrial practice. In retrospect, this transfer provided a lesson in how to, and how not to accomplish technology transfer. The methodology changes required for industrial model checking were achieved through a succession of steps, each of which was small enough to avoid unacceptable disruption of existing methodologies, while significant enough to demonstrate value.

23.1 Introduction

Model checking is an example of engineering at its best, having evolved from an elegant theory to a vital practice. However, any pathway from theory to practice is laced with challenges. Unlike incremental engineering patches that attach directly to current practice, the more revolutionary path must prove that the question it answers is a question that matters.

Engineers have become so good at honing engineering that even when a new discovery presents a means that, in theory, should directly improve current practice, the translation of that discovery to practice must compete against a honed established practice. Thus it was that the sensational breakthrough in linear programming that offered a polynomial algorithm in place of an exponential one nonetheless had a hard time to demonstrate that it actually was faster in practice. Implementations of the exponential algorithm had been so refined that often the polynomial algorithm was not faster on the size of applications that mattered. Examples like this abound. The new, better way must not only compete against years of engineering, it must show that it is enough better in enough important cases to warrant the trouble of replacing the old with the new. Worst of all, there may be no way to demonstrate that the new way is actually better without fully replacing the old way in some context. This is something that the stewards of established practice are loath to do without a

R.P. Kurshan (✉)
New York, NY, USA
e-mail: rkurshan@gmail.com

E.M. Clarke et al. (eds.), *Handbook of Model Checking*,
DOI 10.1007/978-3-319-10575-8_23

prior demonstration that it is better, since any such replacement requires resources that otherwise could be applied to further honing the current practice. This is the vicious circle of technology transfer: transfer requires proof, which requires transfer.

This chapter could be viewed as a companion to [58] where I told of my personal odyssey of verification technology transfer; this chapter tells the story from a general view. I have taken material from the former when the overlap is warranted.

23.1.1 Anything Worth Inventing Is Worth Reinventing

Theories for verifying the correctness of computer programs go back virtually to the beginning of modern programming. These theories were quite varied in their potential application domains and the ways in which they might be used in practice. In one view, computer-aided verification could be said to derive from Russell and Whitehead's *Principia Mathematica* (1910–1913) [85], which laid a foundation for axiomatic reasoning. More germane was Alan Turing's model of computation [79]. The Turing Machine led to automata theory developed by Rabin and Scott [72] for languages of strings and then by Büchi [20] for languages of sequences. The latter provided the basis for automata-theoretic verification [54, 82].

Program verification expressed in terms of a formally defined notion of program *correctness* is widely believed to have started with the work of Robert Floyd [36] (for which, in 1978, he received the first Turing Award that cited advances in program verification). Soon after, Hoare [46] presented a compositional approach to deductive reasoning about program correctness [33]. Nonetheless, Vardi [81] has traced a linear-time form of model checking in which specifications are given in first-order logic or monadic second-order logic to a little known 1957 paper by Alonzo Church [21]. Huuck et al. [47] have observed that Peter Naur [67] developed an approach similar to that of Floyd, one year earlier than Floyd, and pointed out that Herman Goldstine and John von Neumann addressed the issue of program correctness in their 1947 paper on planning and coding for "an electronic computing instrument" [40].

Model checking per se was developed in 1980 by Clarke and Emerson [24] and Queille and Sifakis [71] independently, as every reader of this book knows by now, and its story is wonderfully illuminated in [22, 33, 81].

Earlier, around 1960, computer-aided verification was introduced in the form of automated theorem proving.[1] Theorem proving benefitted especially from the resolution method of Robinson [73], which evolved into other more practical non-resolution methods by Bledsoe and others, leading to the UT Austin school of automated theorem proving that featured Gypsy (by Ron Goode and J.C. Browne) and then the famous Boyer–Moore theorem prover—see [14]. Until 1990, most of the

[1]The term "automated theorem proving", while widely used, was something of a misnomer, perhaps reflecting wishful thinking. These provers tended to be highly interactive. Later, the term evolved into the more accurate "interactive theorem proving" and "automated proof checking".

investment in computer-aided verification went to automated theorem proving. This was especially true for U.S. government support, which was lavished on automated theorem proving during this period [14].

For all its expressiveness and proving potential, automated theorem proving has not quite yet made it into mainstream commercial use for computer-aided verification. One important reason is scalability of use: using an automated theorem prover requires specialized expertise that precludes its broad use in industry. Moreover, even in the hands of experts, using an automated theorem prover tends to take more time than would allow it to be applied broadly and still track the development of a large design project. Therefore, to the extent that automated theorem proving is used on commercial projects, it tends to be used for niche applications like verification of numerical algorithms (hardware multipliers and dividers, especially floating point units, for example). Although AMD and also Intel have made extensive practical use of theorem proving for such applications [50], utilization is presently too narrow and specialized to attract much attention in the Electronic Design Automation (EDA) marketplace, where vendors such as Cadence, Synopsys, Mentor Graphics and others sell software tools for industrial development of integrated circuits. A brief exception was the failed attempt by Abstract Hardware Ltd. of Scotland to market the LAMBDA theorem prover based on HOL [41]. Derived from Milner's ML, HOL had been developed originally for hardware verification. Its commercialization was led by M. Fourman and R. Harris.

Amir Pnueli sought to apply deductive reasoning to verify ongoing computations in concurrent programs, for properties expressed in Temporal Logic [69]. This vision had broad resonance in the field, quickly attracted very many followers and resulted in a major impact that would later garner for Pnueli the Turing Award in 1996 (the second for program verification, following Robert Floyd's award in 1978). Although Pnueli was focused on deductive reasoning, he noted (in passing) that Temporal Logic formulas could be checked algorithmically. In this incidentally observed decision procedure, which later was shown to have non-elementary complexity [81], commercial computer-aided verification could be said to have been conceived. Its birth would come three years later as model checking in the form of a feasible state space exploration algorithm [44]. Ultimately, these ideas would lead to the third Turing Award for work in program verification, given to Edmund M. Clarke, E. Allen Emerson and Joseph Sifakis, in 2007, for the development of model checking.

Analyzing a finite state coordinating system through an exploration of its composite state space was actually proposed by Colin West [74, 84] around 1975. However, it was model checking that formalized the process in terms of checking a model for a property defined in a temporal logic. This difference was very germane to practicality, as it led to decision procedures. Moreover, only with model checking could properties cast as assertions to be checked be reused as constraints to define the environment of an adjoining block. These two, formalization of assertions and algorithmic verification, paved the path to the broad success of model checking.

In fact, the initial gating issues for technology transfer were foremost procedural, not technical. Model-checking theory and technology have continuously far

outstripped its pace of adoption. And this is how we would want it to be: throughout the history of science, the most profound developments followed advances in theory.

This observation imparts a sad reflection on the recent state of "science funding" that requires a demonstration of "need" and "applicability". Indeed, for example, the purely theoretical development of the modal μ-calculus [53] based on recursion and fixed-point operators provided the key, seven years later, to symbolic model checking [63]; there was no "need" and "applicability" for the modal μ-calculus at the time it was developed. Likewise, the Emerson–Lei algorithm [34], which is crucial for the symbolic verification of eventualities, was discovered years before symbolic model checking. In fact, our field happily abounds with examples of theory driving practice: symbolic model checking was based on Bryant's theoretical development of ordered BDDs [18], BDDs having originally been invented by C.Y. Lee [59] and developed by Akers [4] in 1978; Lee's formulation was based on Claude Shannon's work on switching functions [76], which actually harks back to the work of George Boole (1815–1864). Today, we refer to the "Shannon decomposition" for BDDs.

Interpolation [64], one of the most important model-checking techniques of this decade, stems from the basic concept of an "interpolant" whose existence was first proved (for first-order logic) by William Craig in 1957.

Localization reduction (abstraction) [54, 56] or "counter-example-guided refinement" as it has come to be known [25], evolved from an algorithm for the verification of timed automata based on successive approximations [6] (inspired by Newton's method).

23.1.2 The Interplay Between Theory and Application

A test of a field's maturity and robustness is the extent to which its applications "pay back" dividends in the form of new theoretical problems, and cross-fertilization with other fields. If a field does not evolve in concert with new problems raised by its applications, or is insufficiently generic to speak to areas other than those for which it was developed, then that field may not be sufficiently broad, generic or well-formed to last. Model checking has ample evidence of such maturity and robustness.

Perhaps the most stunning example of "payback" is the amazing sequence of advances in SAT solving. Not too long ago, SAT was effective on expressions with fewer than 100 binary variables. Recent advances have increased this by three orders of magnitude. This was stimulated by the demands of hardware design, including especially the renewed interest in SAT spawned by bounded model checking. The impact of these SAT advances on model-checking technology was the impetus for awarding the 2009 CAV Award to the key researchers on the GRASP and Chaff SAT solvers [17].

McMillan's theoretical advances in interpolation [64] were spawned from performance issues in model checking and have in turn produced dramatic improvements

in performance. This interplay continues a dramatic churn between interpolation theory and applications, having generated hundreds of articles in but a few years. McMillan received the 2010 CAV Award [43] in significant part for his stunning insight in applying Craig interpolation to improve the capacity of model checking.

A paper by Rajeev Alur [5] is devoted to examples of such payback. His examples include the invention of timed automata (for which he and David Dill received the first CAV Award in 2008 [45]), recent advances in alternating tree automata and parity games stemming from practical questions about equivalence and synthesis, and new work on push-down automata and his own work on nested words spawned by the needs of software verification.

As a final example, model checking and artificial intelligence have cross-fertilized, with model checking contributing to planning and benefitting from machine learning [33].

23.2 The Technology Transfer Problem

A new technology, by definition, is an alien in the environment of the problems it seeks to solve. In the early years, program verification in general, and model checking in particular, evolved in the gainful rare air of academia, shielded from the grubby realities of practical programming. True, select "industrial examples" such as the Seitz arbiter from Mead and Conway's book [66] served as beacons that rooted theory to practice. But such select examples, while propitiously guiding the evolution of the theory, offered little preparation for dealing with actual programming practice.

Such separation from the sprawling complexity of practice is vital to a developing theory. However, at some point a theory matures, and then we ask "OK, so what is this theory actually worth? What can it genuinely do to help the client?" This is the point of confrontation with the real world, the point where good theories adjust and prevail and those that after all missed the mark fall away to oblivion (or so we might hope...).

23.2.1 Initial Challenges

There are serious theoretical challenges in technology transfer: providing capacity adequate to handle the large size of industrial models, and providing adequate performance to keep pace with the demands of commercial design development. Soon enough inevitable inadequacies here would arouse client reluctance to technology transfer. This reluctance, though, would manifest only once some degree of technology transfer already had occurred. Initially, the most formidable obstacles were not these, but simply coping with the murky details of the client interface, and then resolving how to manage the client's inherent dread of any methodology change.

The first target of model-checking technology transfer was control-intensive hardware designs. This was a natural choice, as explained in Sect. 23.5. For hardware designs, constructing an interface between the model checker and the target meant translating Register Transfer Level (RTL) languages (largely, Verilog and VHDL) to an internal data structure that the model checker could parse. While challenging (for example, typically, multiple assignments to an individual variable had to be collected and represented by a single assignment consistent with all other assignments in the source), this was merely software engineering, for which adequate expertise was readily available. Some useful software was already available to be reused here, for example, elaboration software used for synthesis and simulation; redundant latch removal algorithms that were already in use to optimize synthesis were no less important for model checking, to reduce the number of state bits in a model. Nonetheless, writing a robust, efficient interface for a model checker could take many staff-years—a non-trivial allocation of resources, for which a convincing case needed to be made.

23.2.2 Barriers to Technology Transfer

Of all the obstacles to technology transfer, perhaps the most vexing—because it seems not technologically but emotionally based—is client resistance to change. Anything new is suspect—and for good reason. Just think of all the insightful ideas of questionable practical value that are eagerly advanced by the research community. Industry has neither the bandwidth nor expertise to evaluate them (and engineers eventually tire of hearing about yet another great method that will save the day).

Furthermore, acceptability of a new technology is inversely proportional to the required change in the user interface. A faster compiler that plugs in transparently is an immediate win, because the users see no change (beyond improved performance). Model checking intrudes into the entire development flow. It requires developers to become part of the verification process (by specifying properties). It requires the test team to learn a completely new tool with new concepts and new considerations. The concept of model checking is not transparent to someone who understands testing in terms of executing the design through scenarios. Even today it is sometimes a challenge to wean testers from their tendency to write properties that look like scenarios instead of global properties.

A design factory cannot risk a major disruptive process change that could destroy a thin margin of profitability, even as the change promises to improve profitability in the longer term. It is daunting to assess whether such a change will not only help in a theoretical sense through improved technology, but also help enough in the short term to overcome the inherent costs of process change. Therefore, a major methodology change with its significant attendant costs is almost always a killer for technology transfer.

Of course, change—even somewhat disruptive change—does occur from time to time. In the Electronic Design Automation (EDA) industry, a potentially valuable

change is typically evaluated, first in a dark corner by a summer intern, then, if that shows some promise, by a computer-aided design (CAD) group in their spare time on an old design. Only after a succession of promising evaluations might a new process be evaluated tentatively on an active design (in parallel with the old process). Only when all these evaluation hurdles are passed might the new technology begin to be fully integrated into the design flow.

23.2.3 Methodological Differences and New Potential Benefits

Model checking offered an alternative to the established EDA mode of testing called "simulation test", wherein a design model is exercised by driving it with various input scenarios. The input scenarios generally are generated by a "testbench" that also is instrumented to monitor the ensuing behavior of the design model and detect violations of expected behavior.

The design model is a program written in an RTL language, typically Verilog or VHDL. This same RTL model provides the input to the automated flow from which the resulting physical integrated circuit "chip" is generated, so there is a high level of confidence that functional design failures can be caught through RTL simulation.

A major shortcoming of simulation test is that it inherently is not an exhaustive test. An integrated circuit is a non-terminating device, so any test scenario necessarily is a truncation of an actual behavior. Moreover, the massive degree of parallelism present in integrated circuits leads to more variations of behavior than can possibly be tested in the available time. (The number of variations of behaviors grows exponentially with the number of parallel components that can operate with some degree of independence.) Today, testing is highly automated. Successions of input vectors can be generated by the test bench within 2–3 orders of magnitude of hardware execution speeds by special purpose hardware. Output streams are analyzed automatically and failures, together with testing progress, are registered in elaborate data bases. Such testing may continue for a year or more, running 24 hours a day on server farms—enormous rooms filled with high-speed large-memory computers, for as long as the machines will run, generating potentially trillions of test vectors. Nonetheless, with the massive size of integrated circuit designs that are put into production today, such a testing process can visit no more than a microscopically tiny fraction of one percent of design states. To compensate for this, various often highly sophisticated techniques have been applied to guide these relatively meager test scenarios to the parts of the design that are considered critical. This is mainly accomplished through "guided-random" simulation in which randomly generated inputs are biased so as (hopefully) to exercise the critical scenarios. Such biasing can be adaptive, through monitoring of "coverage" and machine learning.

Nonetheless, critical holes inevitably remain untested. One conceptual problem with this approach is that the problematic areas of a design must be predicted, in order to be tested. This is where model checking has a clear advantage: it can find failures in complex hard-to-imagine scenarios. In fact, many serious design failures

found too late, only once a circuit is cast in silicon or even put into production, are the result of such complex scenarios that slipped through the testing net, never simulated.

As a result, there was a growing sense of urgency regarding the inadequacy of simulation test since at least the year 2000, and the model checking alternative had been discussed in EDA circles since at least 1995 [55]. Nonetheless, introduction of model checking into EDA test flows was excruciatingly slow until around 2005. This was a consequence of the technology transfer problem, especially the required change in methodology.

23.2.4 The Cost of Writing Properties vs. a Test Bench

At first take, writing properties for model checking instead of a test bench for simulation might not seem like a big change in methodology. After all, an automated test bench requires monitors to check for functional failures, and these monitors in effect encode required behavior, as would a property specification for model checking. However, the two modes of expressing behavior are different.

It did not help that test bench monitors, written in an imperative language such as C, often were not conveniently circumscribed, but expressed with various conventions (or no convention at all) and dispersed throughout the test bench code, rendering them difficult to isolate, extract, manipulate, modify and—above all—understand. (In fact, this presented a problem for simulation test as well. This shortcoming provided a serendipitous key to model-checking technology transfer, as it led to support for a unified syntax for simulation monitors that could be used as well for property specifications for model checking. This fascinating story will be revisited in Sect. 23.5.)

Moreover, since simulation is scenario-based, monitors tended to be expressed in terms of required and forbidden sequences of behavior, whereas model checking required a more conceptual specification that could partition every possible sequence into acceptable or not acceptable. For example, instead of writing in a simulator test bench a property that enumerates the various steps of packet-handling under a variety of contingencies—that invariably will be incomplete on account of the combinatorial explosion of possibilities—write the simple high-level property "Every packet is received within two clock cycles after it is sent". (If the actual steps of packet handling need to be checked, these could be spelled out in sub-properties. Most often, though, the "how" is not important.) While this led to much simpler specifications than those attempted by the simulation monitor, some test engineers were uncomfortable with the thought that the model checker could somehow "magically" sort though all the possible scenarios.

Apparently, this only required that test engineers be taught to write their monitors more simply and coherently, and then with model checking they could eliminate the costly test bench altogether. (Writing a test bench is considered a major investment that requires enormous time and effort to get right, so an opportunity to displace this cost would be very attractive.)

Unfortunately, it was not so simple.

23.2.5 Settling on a Scope for Model Checking

Initially, model checking was advanced by the researchers who worked in that field. Their perspective was that in order to reap the full potential benefit of model checking, it must be applied at the system level. At the system level of an entire integrated circuit, model checking could have the greatest impact by verifying system-level properties as understood by the existing product-testing teams. These are the properties that are gleaned directly from product specification documents that describe the high-level functional behavior of the design. In principle, model checking could be made to scale to ever larger designs through the application of compositional verification and abstraction. However, successful applications of such techniques often were tantamount to research projects, requiring considerable expertise and experimentation, not to mention time. (The model-checking problem is PSPACE-complete and thus model-checking decision procedures often cannot be applied directly to very large designs; some form of divide-and-conquer is required and the successful techniques for this require experience.)

It was not long into the model-checking technology transfer odyssey that this mode of utilizing model checking was understood to be a non-starter, scaling neither in time nor available expertise (cf. Sect. 23.2.6). This led to much more modest, scaled-down applications of model checking to design units small enough to be amenable to direct application of the decision procedures. In this mode of use, capacity limitations limited model checking to a small number of design blocks (often one or two). This mode of verification stood in stark contrast to the mode familiar to the product test teams, in which the entire design was simulated all together. It became a requirement for the application of model checking.

This requirement had two very disruptive consequences: properties had to be written for individual blocks, and constraints had to be written to model the environment of each block. This work naturally fell to a Verification Team. However, although such teams understood the design behavior at the system level, it was difficult or impossible for them to acquire the required block-level knowledge within the time and resources allowed to them. Block-level knowledge frequently resided only with the designers, who could not or would not be bothered to provide this information. (More on this below. In some cases, the designers had moved on to other projects or even other companies—a situation that also held grave consequences when debugging was required, by the way.)

The preferred solution to this problem was to move model checking "upstream" in the design flow to the designers themselves. Optionally, the verification team could be re-educated to work with the designers at the block level. This solution has some very compelling advantages. It puts block-level verification in the hands of the person who best understands the required block-level behavior: the designer. Moreover, it gives the designer a very powerful debugging tool: a model checker. Without this, a designer may merely check that completed code compiles, since writing a test bench for a block typically is considered prohibitively costly. Without block-level functional verification, functional bugs can only be caught much later, during the system verification stage of the development flow, when fixing bugs is

much costlier. There is a widely cited sense in the industry that for each development stage in which a bug's discovery has been deferred, the cost to fix that bug has been increased by an order of magnitude. ("A stitch in time saves nine.")

Although this presents a very compelling argument for moving functional verification upstream to the designer and using model checking, this move has been considered extremely disruptive for at least two reasons. Designers are considered too valuable to divert to verification. (Classically verification is considered to be a less-prestigious activity, although today design projects are sinking under the weight of verification, which has been estimated to consume 50% to 80% of total design time.[2]) Requiring verification engineers to perform block-level model checking in collaboration with designers is thought at best to require time-consuming retraining, and at worst to require an engineer with different qualifications altogether (entailing laying off existing verification engineers and recruiting others).

Groups that have nonetheless experimented with this soon discovered an even more daunting obstacle. To perform block-level model checking, it is not enough to have an understanding of the functional behavior of the block in question (which in principle could be obtained from the designer of that block). It is also necessary to have an understanding of the functionality of adjacent blocks, in order to write an appropriate environment model as required by model checking. The best way to do this is to write constraints on block inputs, as then the constraints can later be reused as assertions to be verified on the adjacent blocks. To do this properly requires a mechanism to syntactically transform a constraint to an assertion by transforming inputs to outputs. While this may seem trivial (and it generally is—at worst, there is only a bit of renaming ugliness to overcome), it is not commonly supported in today's model-checking tools, in which case such transformations must be done manually (although, see [51]). At the same time, there would need to be checks against the potential for circularity in this assume-guarantee setup. Again, while this is not hard to provide, it is not commonly provided today, and so must be handled manually.

The reason that such seemingly simple transformations and checks are missing from many commercial model checkers is not on account of oversight. It is the result of another vicious circle in the technology transfer saga. Features and enhancements to a product get implemented in a priority order. When development resources are scarce, important priorities can repeatedly get displaced by other more urgent priorities (such as fixing bugs and responding to critical customer issues). However, the inability to deliver such important priorities adversely impacts the attractiveness of the model checker and thus the sales revenues it can generate. Reduced sales revenues often lead to reduced resources for product development. While this vicious circle can be broken by enlightened management—and virtually every commercial model checker is testimony to an example of enlightened management—such enlightenment is very vulnerable when the focus is on near-term returns and cost-saving (especially in recessionary times).

[2]This widely cited estimate was confirmed by the independent 2010 Wilson Research Group Functional Verification Study described in a blog by Harry Foster [38].

23.2.6 The First Commercial Successes

In fact, model-checking technology emerged into the commercial arena mainly through start-ups, where explicit funding could be secured to break this vicious circle. Successful examples of these were Verplex (acquired by Cadence), 0-In (acquired by Mentor Graphics) and, most recently, Jasper Design Automation (acquired by Cadence in 2014). A few commercially successful model checkers were produced in incubators in larger organizations, including FormalCheck (acquired by Cadence from Lucent in 1998), and RuleBase and SixthSense developed by IBM (mainly—but not exclusively—for internal use). Many other model checkers were developed both in start-ups and larger organizations, but, for one reason or another, failed to be commercialized.

A singular exception to this start-up/incubator rule was Intel. Intel was possibly the first design company to seriously support model checking, having begun their effort in 1990 [58, 81], and continuing it successfully and impressively until today. That they could break through this vicious circle is probably mostly due to their sense of urgency to cope with the verification problem, coupled with enormous resources. The verification problem of how to deliver ever more complex designs that are functionally correct undoubtedly hit Intel first and hardest, and they have definitely been a foremost leader in the development of model checking for commercial use. With the exception of Intel and IBM, design companies prefer to acquire model-checking technology from EDA vendors. (And even Intel and IBM have recently augmented their internal model-checking capabilities with EDA vendor tools, perhaps attracted by the potentially greater economies of scale afforded to EDA vendors that should enable them to deliver less expensive and better technology.)

A number of other circuit design companies attempted and then abandoned the development of a model checker, ultimately preferring to acquire the technology from EDA vendors such as Cadence, Mentor and Synopsys. Design companies also opened themselves to model checkers from start-ups, sometimes by way of evaluating the model checkers from the larger EDA vendors. Often, a design company ultimately prefers to get its tools from the large EDA vendors, for reasons of reliability, service, long-term relationships, tool interoperability and the opportunity for package deals. This preference rarely stood in the way, however, of the design companies licensing the start-up products to "encourage" the larger EDA vendors to improve their own competitive products. In the end, it works out well for a spectrum of winners: the best start-ups get acquired by the major EDA vendors and the design companies drive steady improvement of the products. A stunning exception to this pattern was Jasper, which remained an exceptionally successful start-up for 15 years before it was finally acquired by Cadence in 2014. During its start-up period, it had evolved into a leading vendor of model-checking technology.

23.2.7 The Essentiality of Being Able to Compare Tools

A client's ability to compare the variety of model-checking tools available is an important precursor to a decision to purchase. In the early days of commercial model checkers, each checker was based on its own property specification language. This meant that in order to compare several model checkers on one design, the design specification would need to be rewritten for each tool. Comparing competing model checkers under such conditions was beyond the wherewithal (or patience) of the client design companies, and led to a general reluctance to go beyond experimental usage of one tool or another. While initially experimental usage of a vendor's product raised that vendor's hope of an eventual big sale in which the tool would be broadly integrated into the client's development flow, the clients were largely and understandably reluctant, as they felt that they had not been able to explore all the competitive options. This situation remained a barrier to widespread client acquisition of model checkers from the EDA vendors until this barrier was overcome by the establishment of an international standard for property specifications [81]. The urgency for such a standard was so great that, once promised, before the ink was dry, the EDA vendors were implementing the standard in their tools. With a standard for property specification supported by all the commercial tools, the client companies could evaluate all the available model checkers on one and the same set of designs and property specifications, finally opening the door to serious acquisition and proliferation of model checking within the client companies. This in turn emboldened vendors to develop commercial model checkers. The ability of clients to compare the vendors' model checkers head-to-head provided a great stimulus to technology transfer and revenues from model checker sales. Indeed, competition among vendors breeds confidence. Several vendors advancing similar products lends verisimilitude to the field, whereas a unique product without competition is suspect. Today, every major EDA vendor markets a model checker. Although competition is sometimes feared, in this case it increased every vendor's sales and was a requirement for widespread acceptance of the product.

Still, capacity (the size model that can be checked) has always been an issue. The smaller the capacity, the more partitioning is required for verification. Partitioning at RTL block boundaries is natural (given that partitioning is necessary). If capacity is below the size of a typical RTL block, then the block itself must be partitioned. Typically, partitioning a single block would be considered too much effort and too fraught with the risk of introducing errors to be seriously supported in a real development flow. Therefore, block-level capacity is an essential ingredient for technology transfer. The advent of BDD-based algorithms to support symbolic model checking [63] elevated model-checking capacity above block level, thus eliminating capacity as a show-stopper in many cases. Since then capacity has steadily improved, especially with the introduction of SAT-based symbolic model checking [23, 64], which has afforded as much as two orders of magnitude increase in capacity over BDDs.

As clients began to seriously evaluate competing model checkers, they became aware of a plethora of confusing issues, details and use models. In some tools,

considerations driven by capacity limitations require some (usually limited) understanding of abstraction, partitioning, compositional verification and restriction. Even when these are automated in the tool, the boundaries of the automation can be reached and then must be addressed. Since performance is critical, the user often must become familiar with a variety of "engines" such as BDD-based algorithms and SAT-based algorithms, or complete algorithms for verification vs. incomplete falsification engines such as bounded model checking [23]. These have different sweet spots that may be hard to identify automatically. While such issues did not present barriers to technology transfer, they did present hurdles that slowed acceptance of model-checking technology. Vendors were required to address these issues. The additional requirements on tool development came together with client reluctance to proliferate model checking until the issues were managed. This retarded the vendors' anticipated revenue growth and had an unfortunate damping effect on the rate of technology transfer.

23.2.8 In Summary

All in all, model-checking clients have complained (with some justification) that although model checking came with a promise to eliminate the costly task of test bench development, it was accompanied by the also costly task of writing an environment model, not to mention all these other growing pains associated with the adoption of a new tool, let alone a new methodology. That writing an environment model is actually considerably simpler and less costly than writing a test bench, especially given the reuse of the environmental constraints as assertions for the adjacent blocks, somehow gets lost in the translation. (The clients are justified, though, in that support for this typically is inadequate, and in any case would require additional training.)

And yet, notwithstanding all these impediments to technology transfer, there have now been many examples of successful model-checking technology transfers in the last five to ten years. How these transfers first were (unsuccessfully) attempted and then finally succeeded are the topics of Sects. 23.3 and 23.4.

To successfully transfer a disruptive technology as described here entails a simultaneous solution to five challenges:

1. break the vicious circle of funding (transfer requires funding that requires proof that requires transfer);
2. interface the new technology to the client environment;
3. limit and then integrate methodology changes into client practice;
4. demonstrate the cost-effectiveness of the new technology in the client's environment;
5. enable competitive evaluations.

In summary, it takes much time and thus cost to generate confidence in a new disruptive technology. While it is vitally important that factors such as language

standards are present to ease the process, these are only necessary but not sufficient. In view of the high cost of any significant change in methodology, technology transfer requires a compelling need in order to be cost-effective. In fact, as explained in Sects. 23.3 and 23.4, the industry both waited for the need to overcome the cost and at the same time took steps to reduce that cost.

23.3 False Starts

In the early days of program verification there were seemingly endless arguments over which was the best modeling paradigm to support program verification—CCS? CSP? branching time? linear time? All these late-night fulminations turned out to be mostly misguided: the choice of modeling syntax or even semantics was not in fact part of the problem; for better or worse (and many would argue, mostly worse) the choice of programming language was already cast in stone and could not be changed. Countless hours invested in developing myriad design model formulations honed for concurrency and verification, with one or another particular partisan bent, at best led to academic clarifications of the issues at hand. Eventually, the stark truth struck: we are stuck with a few RTL programming languages for hardware models (mainly Verilog and VHDL) and a few flavors of C for software models. While new programming languages obviously do get born on rare occasion, the ontogeny is generally obscure. But there was a vibrant supply of purists who did not care: they insisted on leaving their mark in the dust bin of recorded literature, registering their take on the perfect language for verification. The alternative pragmatic view was that it is better to figure out how to verify programs written in the languages that fate dealt. Once this view became dominant, our community buckled down in the late 1980s to the first hard task of coupling verification algorithms to an existing infrastructure of compilers and elaborators.

This is not to say that the development of low-level languages like SMV was misguided; on the contrary, these provided a vital simple-to-understand interface, and what is more, imparted a semantics to the (semantics-free!) languages in use. These low-level languages were the “assembly languages” of formal analysis. They were needed to capture the essential elements useful for verification, like a single-assignment syntax from which a transition structure could easily be derived, declarations for property specification, and other special structures that could be exploited to simplify model checking, like the explicit identification of state. At the same time, the low-level language needed to be suitable for translation from the languages in use. While the simplicity and clarity of these low-level languages did gain them a following who advocated for their use as design languages, such use never flourished. The reasons for this may in part be on account of their spartan syntax (a benefit, in some minds), but mostly it was simply that the established languages already commanded too many interfaces with various applications to be candidates for replacement. This common situation should provide a reality check for every grand scheme architect.

The most distracting false starts in the transfer of model-checking technology from theory to practice all could be characterized with a single noun: *over-ambition*.

First, there was the attempt to position model checking at the system level, where it can confer the greatest benefit by moving verification upstream in the design flow to catch bugs earlier, scale with design size (by using the compositional techniques anyway required at that level) and address the most important (system-level) properties of the design. At the time, this positioning of model checking seemed very reasonable, but as already discussed in the previous section, this was a non-starter on account of the immense methodological change it would entail and its dependency on advanced expertise. (In fact, system level model checking is the ultimate goal for all our EDA tools, although not viable as a first step.)

The path from the first widely disseminated commercial model checker FormalCheck [37] released in 1997 to today illuminates a succession of decisions to first replace power and flexibility by automation and ease of use—and then reintroduce power and flexibility as users became more sophisticated. At first, users wanted something that was simple and easy to use, requiring little understanding of the concepts. As users became more comfortable with model checking, they demanded greater power and flexibility, and, ironically, sometimes these were provided merely by turning features back on that had been implemented years earlier and then turned off in the interest of simplicity. Today, conceptual simplicity in commercial tools is becoming less of a concern; there are enough expert commercial users to support the development of many advanced options and warrant having all such implemented features available, at least as hidden options.

Sometimes the most immediately useful part of a new theory is a trivial part considered hardly worth the mention. This accurately described the evolution of commercial model checking. (As a parallel, Emerson [33] has described how model checking itself came out of his study of the much harder program synthesis problem.)

Compared with "theorem proving" (automated proof checking), model checking could be considered much simpler—certainly from the perspective of the user. This accounts for the much wider proliferation of model checking in EDA, compared with theorem proving.

Whereas early emphasis in model checking was placed on *verification*, it did not take long to realize that as a first step, a tool that focused on *falsification* was more accessible and immediately useful to hardware verification groups.

Before the wide spread acceptance of formal verification, one could go to a designer with the claim "I verified that your design is correct" and you might get a muffled yawn, because "who knows what this claim of verified really means". But, show her a bug in a form she could readily comprehend (a simulation error trace) and she could immediately recognize the value: you found a bug that she did not know was there. From the first applications of model checking, it was clear that the first real value of model checking was that it was an uncommonly good debugger—*reductio ad bug*. It was more effective to find a bug by trying to prove that there were none, than to go looking for it directly—the bug is always where you didn't think to look. As a debugger, the model checker was forgiven for any black magic that went

into its application (like abstractions, restrictions, unsoundness) since it was only the bug that mattered; no need to understand the means. “I don’t need to understand what model checking is, because I know a bug when I see it (and ‘verified’ only means ‘failed to find a bug’).”

With the focus on falsification, completeness became less of an issue. This opened the door for under-approximations such as restrictions on inputs (for example, setting some data paths to near-constants) and restriction of the depth of search as with bounded model checking.

Even soundness became expendable to some extent. This sounds horrific, but even in the early days, soundness was discarded when error tracks from abstractions were admitted. Likewise, if a block has been checked, but its environment model has not been verified, then one cannot be sure of the soundness of the result. Often, environment models are correct-by-construction abstractions of the actual environment. Then, however, property failure in a block does not necessarily imply a failure in the full design (in the full design, the actual block environment may be more constrained). The user would need to try to simulate the error track. As long as there were not too many false fails, the user was content. At no time would a false *pass* be considered acceptable, and yet almost no product is without bugs, so there may well be some false passes out there—if only as an artifact of faulty bookkeeping. But this is the real world: you do your best, but keep moving. A Herculean effort to verify the correctness of a model checker would not be a welcome diversion of resources. Early focus on proving the soundness of the model checker was misplaced. More important than soundness is coverage: instead of worrying about correctness, worry about writing more properties to check more of the design. That’s the value assessment. After all, simulation test has known semantic faults [35], and these are tolerated. Only academics worry about them.

Today, all this is changing. Commercial users understand what “verified” means, and there is an increasing emphasis on semantic correctness and completeness of the verification enterprise. In other words, commercial use has converged to the use intended by the academic community decades ago.

As the developers of commercial model checkers became increasingly in tune with their clients, tool interoperability became increasingly important. In the early days, model checking was an alternative to simulation test. The two at best were disdainful neighbors: model checking could not put simulation out of business, although the secret wish was there. Simulation was the workhorse of the test community; model checking by comparison seemed a toy. As a result, the two efforts evolved in different groups. The company 0-In (recently acquired by Mentor Graphics) may have been the first to establish peace between the two in the form of an alliance aptly called *hybrid* verification (not to be confused with “hybrid systems”). The idea was to use a simulator to quickly drive the design to “deep” states unreachable with a model checker. From those deep states, model checking (perhaps on an abstraction) could be started. Synopsys has perfected the idea with their tool Magellan, Cadence has their hybrid checker IEV, and today others are finding other useful ways to combine simulation test and model checking. Certainly, a simulator is handy for validating an error track on the full design. A big virtue of this hybrid

is that it moves model checking into more familiar territory for the vast majority of the testing community who practice simulation test. Through this connection lies the best chance to put a model checker on the desk of everyone who today runs a simulator. The idea of hybrid verification was overlooked (or passed by) because it seemed to those working in model checking like a meager contribution. In retrospect, it was an important early opportunity that was overlooked. A small but valuable step.

The most immediately successful mode of model checking has been logic equivalence checking: verifying that two net-lists define the same Boolean function. While equivalence checking evolved separately from full model checking of sequential designs, its rapid rise to success in the EDA industry should not be overlooked as an example of a conceptual idea that's very simple, enjoying broad use (and producing very significant revenue streams) very quickly. It did not suffer from the principal barriers to technology transfer because it fit into current methodology. But is it a good example of the power of a simple idea.

23.4 A Framework for Technology Transfer

Despite the formidable barriers to the transfer of model-checking technology to industrial use, those barriers eventually were breached. How did this happen? The answer is simple, although the process was not. Technology transfer was achieved through a succession of small, incremental steps, each of which moved in the direction of industrial adoption and collectively, over more than a decade, achieved that goal. Through small, incremental steps, excessive disruption of existing industrial design development flows was avoided. However, to be worth the effort of adoption, each small step nonetheless needed to offer some benefit over the current practice. The cut point is cost-effectiveness: the small step needs only to provide a short-term benefit greater than its adoption cost. (Longer-term benefits are too hard to predict and thus are generally heavily discounted.)

It would be nice to report that at some point those of us involved in the technology transfer process were smart enough to understand this picture and to drive technology transfer by its rules. However, even if there were those who understood, the principle informs but fails to guide. Which small steps comprise the right succession to repeatedly, step after step, provide the required cost-effectiveness and lead to the desired goal? The truth is that we blundered into the solution (and indeed continue to do so, as this technology transfer will be ongoing for as long as model-checking technology continues to evolve, and practice continues to trail behind available technology). Trial and error through a process of "natural selection" led to discovering the succession of cost-effective incremental technology transfer steps that got us to where we are today.

A Grand Architect of technology transfer could come up with a "road map" of incremental steps that get from where we are today to full adoption of model-checking technology—solving the "getting from here to there" problem. That would take

great vision in view of the many course changes that actually occurred in the transfer of this technology. While such a road map is an attractive ideal, reality manages with blundering forward.

Around 1990, major IC (integrated circuit) designers like Intel, IBM and the cellular telephone manufacturers, along with the automotive and avionics industries, began to understand that the increasing functional complexity of designs was rendering them untestable. Without a dramatic change, decreasing reliability would become the gating factor for design complexity. The new circuits could be built, but they would malfunction. There is nothing like a half-billion dollar bug [28] to drive this message home. Furthermore, evidence suggests that Toyota's recent braking problems may be linked to a "software glitch" [3, 9] whose cost could dwarf the Intel bug: in the two months following the brake-related recalls of January 2010, $25 billion of Toyota's market capitalization was wiped out [48]. While ongoing computer security bugs are rampant, pervasive and extraordinarily costly ([83] estimates a loss exceeding $50 billion in 2006 alone), these are unlikely to be found through model checking: to think of the model in which the security breach can occur is to already finger the problem. (The vastness of the Internet with its entirety of hardware and software certainly precludes modeling, much less verifying.) Yet there were other bugs, infamous for the high cost of the damage they wrought, that arguably could have been discovered though model checking. Among these are: the deadly Therac-25 race bug [60], the 1996 $370 million loss of the Ariane 5 rocket [61], NASA's 1999 $125 million loss of its Mars Climate Orbiter [27], and the incalculable losses from AT&T's 1990 voice network collapse [77, 78] and 1998 data network collapse [13, 62]. All these were soon enough traced to hardware and software bugs that while easy to understand after the fact, were too hard to imagine and thus discover through conventional scenario-based testing. Alongside this trail of sensational bugs lies the litter of the mundane bugs that have degraded performance, increased costs and have been shown after the fact to have been discoverable through model checking; and along with these are the bugs that could have degraded performance and increased costs, but were discovered by model checking after having been missed by simulation test.

The result today is a rapidly growing sector of the IC design community that has come to realize that maintaining testing-as-usual would drive them out of business on account of the rapidly degraded quality that accompanies increasingly untestable functional complexity. They are driven to seek new remedies, including guided-random simulation mediated by learning, and model checking. This increased need finally opened the door to model checking in EDA—just as the academic community had been predicting for decades. But the EDA industry yet needed to come up with a cost-effective way to introduce the technology. The trial and error process that accomplished this was fraught with far more issues than the academic technology innovators ever imagined.

Equivalence checking, the first major success of model-checking technology transfer, is such a special case of model checking that it generally is not even considered as a transfer of model-checking technology. However, it remains a perfect example of a small step in the grand plan, and it definitely opened the way for more

general forms of model checking. Equivalence checking grew quietly in the 1990s before it burst forth as a very successful product that is used extensively and broadly in the EDA design development flow. The ideas of equivalence checking had been simmering quietly since around 1980 [19]. Back then the problem was capacity, but the main ideas—mapping states to convert sequential equivalence checking into combinational equivalence checking and then further mapping nets via names and topology—were already established. With the advent of BDDs, the capacity issue was ameliorated and much better products emerged. The research community took little note, as the great challenges were in general model checking. But suddenly, by the turn of the century, equivalence checking was fully established as a mainstream commercial product. A small step, but one of great significance and value. Clients of general model checking were less intimidated by its algorithms once they had become experienced with those algorithms in the framework of equivalence checking and had learned how valuable they could be. Even today, "formal verification" means equivalence checking to many clients of the EDA industry.

IBM's equivalence checker Verity developed by Andreas Kuehlmann in 1992 was perhaps the earliest modern (BDD-based) equivalence checker. In 2004 Verity was sold to Magma, which now markets it as QuartzFormal. Since then other leading equivalence checkers came into being: Cadence's Conformal (acquired along with Verplex), Synopsys' Formality, Calypto's SLEC, IBM's SixthSense and FormalPro from Mentor Graphics.

The biggest incremental opening for model checking in the course of its technology transfer odyssey came as unexpectedly as it was in plain view. This opening was created by the establishment in 2003 of an international standard for functional property specification languages, the Accellera Standard [2]. (Unfortunately, on account of commercial pressures, the standard established not one but three alternative languages: OVL, technically, a library of assertion checkers, PSL and SVA, all of which EDA vendors were obliged to support.) This standard allowed for head-to-head evaluations of competing commercial model checkers, and opened the way for serious sales and proliferation of model checking technology, as already explained. Interestingly, while the basis of this standard came from model-checking specification languages, the driving motivation behind the standardization had little to do with model checking. The motivation came from the parallel needs for clarity, modularity and standardization of monitors used with simulation test benches. This evolved as a program known as "Assertion-Based Verification" (ABV) [39], wherein monitors evolved into "assertions". This was the start of a beautifully symbiotic relationship between model checking and simulation test. The idea of a functional property specification language based on assertions originated with model checking; it was popularized and standardized for simulation monitors, and then it was returned to model checking for commercial use there as well. Thus, model checking served simulation by providing a way to clean up and standardize simulation monitors; simulation provided the muscle to actually accomplish the standardization and technology transfer; and then model checking consumed the resulting property specification languages to surmount a technology transfer obstacle that hitherto had blocked the possibility of serious commercial proliferation of model checking (as

described earlier). I do not believe that anyone who advocated for model-checking technology transfer ever predicted the specification language obstacle that resulted from its lack of standardization, much less this stunningly serendipitous solution, until the standardization process already was under way.

The symbiotic relationship between model checking and simulation test continues to flourish. Combining model checking and simulation for "hybrid" tools is now rapidly gaining use (after a slow start) as the interoperability between the two improves. The potential for such synergy may not have been predicted in the early days when the prevailing sentiment among some academics envisioned model checking as a replacement for simulation. (That this sentiment was quickly repressed in order to avoid alienating potential EDA clients does not mean that it was not nonetheless held.)

Most recently, simulation test has benefited from BDD- and SAT-based constraint solvers used for guided-random simulation. These solvers come directly out of core model checker technology.

Today, there is a robust respect for formal methods in the EDA community and model-checking technology is so thoroughly integrated in the verification flow that it is no longer an alien technology seeking to gain a foothold, but just part of the product testing arsenal. Nonetheless, the technology transfer gap is far from closed. The next big technology transfer challenge for model checking undoubtedly is system-level verification: how to transfer current technology and techniques for system-level model checking to commercial practice. There are ideas how to do this, and behind these ideas, pilot projects. I believe that our recent odyssey has demonstrated that to breach the barrier that this new challenge presents will require a succession of small, non-disruptively incremental and yet cost-effective steps. I think history has demonstrated that technology never stands still, and the increasingly critical need for system-level model checking will find its way. And I think that experience has shown that we are unlikely to correctly predict the road map of incremental steps that will get us there.

23.5 Formal Functional Verification in Commercial Use Today

Equivalence checking is broadly used and of all formal methods is the one that has most pervasively penetrated the IC industry development flow. An initial circuit design is subject to various optimizations like retiming, clock-gating (pruning the clock tree to reduce dynamic power dissipation when a flip-flow has a don't-care value) and similarly, at a block level, power-gating (turning off blocks of a design not in use, to save power). These optimizations, even when automatic (algorithmic), cannot always be assumed to be correct by construction. The original design is typically easier to understand than the transformed design, and may have already undergone testing. Therefore it must be checked that the optimized design is functionally equivalent to the original design. Functional equivalence checking is thus an essential and central activity in EDA.

To verify that an optimized sequential design is functionally equivalent to its original (unoptimized) design, one tries first to decrease the computational burden by reducing the problem to a series of combinational equivalence checks—checking the equivalence of stateless Boolean functions. This can be accomplished through various techniques that associate the states of the two designs. Checking the equivalence of stateless Boolean functions can often then be further simplified through associations of logic nets that may remain untransformed between the two designs or are equivalent at certain states of the original design.

Presently, the focus in EDA is on combinational equivalence checking, as most commercial circuit equivalence checks have been thus reducible to combinational checks. However, the remaining parts of the check that cannot be reduced are handled as (sequential) model-checking problems and these seem to be growing along with an increasing complexity of transformations that defy simple reductions to combinational equivalence checking. While, organizationally, equivalence checking and model checking have developed separately in the EDA industry, increasing dependence on full sequential equivalence checking incorporating model checking seems like it will inevitably bring them together.

Full functional verification of hardware through model checking—at long last—is now a success story of technology transfer of computer-aided verification. As recently as 2005, the jury was still out on whether the then nascent commercial hardware model checking would gain traction in the EDA industry. The barriers to technology transfer were primarily procedural as already explained, and the technology had far outstripped its supported level in EDA. Today, commercially supported EDA model checking is mostly for local (RTL block-level) properties. These include properties such as arbitration, resource allocation (request/grant properties), flow control (underflow/overflow), state unreachability, local message delivery, and local serialization, as well as an endless list of hardware particulars (sometimes of questionable value, having been derived from a simulation mindset rather than the mindset of correct functional behavior). Very recently, we begin to see EDA application of model checking to system-level properties, a very welcome sign. Often this requires some model-checking expertise to push through.

The final great enabler of commercial hardware model checking was the introduction of ABV based upon the Accellera Standard for functional property specification mentioned in the previous section. Thus, two great enablers of model-checking technology transfer were provided by established technologies: synthesis optimizations (Sect. 23.2.1) for state-efficient models, and ABV for standardized functional property specification. The importance of these for technology transfer—and the good luck of the synergies for model checking—are often overlooked. Software model checking will need both of these: efficient state models and a standardized functional property specification language before it can become a broadly supported commercial technology, it would seem.

With EDA providers supporting the Accellera standards, CAD groups could build their flow for model checking without being tied to a specific product. This liberation was a prime enabler for the acceptance of model checking in mainstream design flows. The interoperability afforded by the standards bred competition among

the vendors, predictably increasing the quality of the model-checking products. It also liberated start-ups from the need to create a functional property specification infrastructure, allowing them to focus on the core technology. This in turn bred more competition.

Competition breeds confidence: confidence for the consumer that they can switch to the best available product without disturbing their flow; confidence for the vendor that this is a product of significant value.

Today there is an impressive array of EDA vendors competing in the model-checking arena, including the big three: Synopsys, Cadence and Mentor Graphics, as well as a swell of start-ups with impressive model-checking products, especially Jasper (recently, no longer a start-up), as well as Calypto, Prover, Averant, RealIntent, @HDL, and others. Although not really an EDA vendor, IBM has nonetheless made impressive headway marketing their RuleBase model checker as well. However, this landscape is constantly moving, and by the time this is published, it may well be out of date.

While not yet widely supported by the EDA industry, model checking high-level (abstract) models of complex protocols such as cache coherence is routine in the processor design industry and support for this is only just now starting to make its way into commercially available EDA model checkers. However, at the time of writing, this application of model checking is less mainstream than checking local properties of RTL blocks, despite its importance and perfect fit with model-checking technology. Meanwhile, the tool of preference for those who design cache coherence protocols has been Murφ [31] designed by David Dill and his students. While David has moved on to other interests, Murφ is actively maintained by Ganesh Gopalakrishnan at the University of Utah, thus providing an important service to this segment of the industry until the EDA industry catches up.

Interactive theorem proving is used today in a few niche applications, primarily for data-path verification of numerical algorithms such as multipliers. AMD and Intel have found a way to use this effectively in their development flow [50]. Today it is not supported commercially by the EDA industry (although in the 1990s it briefly was, in the UK). The reasons for this have already been discussed.

The application of formal methods to software design and development has proved much harder than for hardware.[3] There are numerous reasons for this. First is the relative lack of semantics for C code (cf. [11] for a penetrating and sadly humorous reflection on this). Hardware specification languages were no better, but had a semantics forced on them by the need to automatically synthesize a program into a circuit. It is precisely this imputed semantics that provides the hooks for model

[3] In our context, "hardware" means HDL, e.g., Verilog code, whereas "software" means primarily C or C++ code. Of course, pedantically speaking they are both "software", whereas "hardware" is something attached with a soldering iron. The contrast between "hardware" and "software" becomes more blurred with the advent of "system-level" design languages such as SystemC: with restricted C syntax, they compile directly into an RTL language. Moreover, there have been cases of successfully model checking restricted forms of C [8]. Here, when speaking broadly of "verifying software", what is meant is verifying the full syntax of C, including, most problematically, pointers and memory allocation.

checking. Formal analysis of software entails assigning a semantics that in some cases may be arbitrary and fails to match the semantics imputed in other contexts.

Moreover, the performance of synthesis for hardware is intimately tied to an efficient assignment of sequential elements (states) to the design. Optimizing which program variables are "state" variables has been the object of enormous effort in the EDA industry for the purpose of efficient synthesis. Model checking capitalizes on this effort by using the synthesis assignment.

Since software has no such concern for synthesis, model checking for software must incorporate a step wherein sequentiality is imputed to the program: which actions can be assumed to happen simultaneously, which must occur sequentially. Again, doing this efficiently is very important for the performance of the model checking, but in the case of software, there are no years of effort in this direction to stand upon. The general solution for software has been to assume that *all* actions are sequential, and that actions in different "processes" interleave. This is satisfactory for small software models, but does not come close to passing the required component capacity threshold needed for model checking discussed earlier. Interleaving also tends to be more efficiently implemented by explicit state analysis, further limiting capacity.

There are still other impediments to software model checking. Since C is the language of choice for designs, pointers, memory handling and the general infinite state of the system all must be handled. While there has been some impressive progress in this area [26], it has some distance to go before it can be picked up for commercial use.

Nonetheless, there has been one important achievement in commercial software model checking: at Microsoft, SLAM [8] has penetrated mainstream driver development (for which its principal developers, Thomas Ball and Sriram Rajamani, were awarded the 2011 CAV Award [80]). It uses a software model based on a driver program's control flow graph to overcome all the problems cited above. Its verification is based on a push-down automaton associated with the control flow graph. While SLAM has been fanned out into the design development process in Microsoft and is an important example of model-checking technology transfer, its very particular application must be understood as brushing the boundary of what is currently possible for model-checking software in an industrial setting.

There have also been some very impressive recent but by now fairly widely used advances in commercial static analysis tools for C and C++ code (and in at least one case, Ada). See [32] for a comparative study of a number of these. These commercial static analyzers include Coverity [30], Klocwork [52], Polyspace [70] acquired by MathWorks, Parasoft [68], Astree [7] from AbsInt Angewandte Informatik [1] based upon the theory of abstract interpretation developed by the Cousots [29] and, most recently, CodeSonar [42] from Grammatech by Tim Teitelbaum and Tom Reps from the CAV community.

Europe has recently taken a more "progressive" attitude toward technology than the US or the Far East. (I attribute this to the pervasive availability–at least prior to the current economic crisis–of government and now EU funding for new technology, allowing many ideas to flourish without passing the acid test of the free market, as

required elsewhere. This atmosphere is reminiscent of the lavish post-Sputnik funding in the US, fueled by freely flowing government grants in the sciences, defining a period that coincided with the "golden age" of research in the US, but also with a level of "dead-end" research that would no longer be tolerated.) This also may be reflected in the observation that EU applications tend to be more "pure", compared to a US focus on the more practical. For example, of the static analyzers listed above, Polyspace and Astree from the EU are more focused on verification, whereas the others are focused on falsification (bug-hunting). Be that as it may, there are a number of commercial formal verification successes in the EU. In addition to Polyspace and Astree, there was Esterel Technologies whose web site had many pointers to its industrial applications. Although model checking was not a key to its success, the ability to analyze formal models may have been. The background technology was developed over more than a decade by Gérard Berry [10]. Esterel Technologies has applied formal analysis to high-level models of critical applications, prominently aerospace and automotive, thereby circumventing the obstacles of C code verification.

Likewise, iLogix (acquired by Telelogic, which in turn was acquired by IBM) has seen commercial success with its implementations of StateCharts. Driven by David Harel and Amir Pnueli, the basic ideas have evolved in the literature over decades. Again, the object is analysis of high-level models. iLogix has had successful use in the telephony industry. Again, while based on a formal semantics that is important for the analysis they do, formal verification may have been peripheral to the company's success.

23.6 Algorithms

Until the advent of symbolic model checking based on BDDs [63], model checking was based on explicit state enumeration. This typically allowed a few million states to be searched, a bit more today with bigger and faster computers, but not enough to provide the capacity required to break through the RTL block threshold described above. With BDDs, blocks with 10^{50} states or more could be checked. This corresponds to 100–200 state bits, in contrast to around 20 state bits for explicit search.

However, BDDs have their own problems, mainly their often mathematically chaotic and thus unpredictable memory requirements (changing the location in the design of a single variable declaration can result in an enormous and generally unpredictable change in the size of the resulting BDD). In the late 1990s Ed Clarke and his co-authors suggested using SAT as an alternative for model checking [12, 23]. (Some, including this writer, voiced skepticism regarding the value of this approach, on account of the then very limited capacity of SAT solvers, and in time we were proved wrong. Ed and his co-authors deserve great credit for their foresight.)

Today, model checking can be based on explicit state enumeration or symbolic state enumeration based on BDDs or SAT, as well as ATPG (Automatic Test Pattern Generation). Each of these has given rise to a number of distinct model-checking algorithms (see below). McMillan's interpolation method [64] was the first complete

SAT-based model-checking algorithm, and on our benchmark tests in Cadence this generally outperformed all the others for a number of years. More recently, Aaron Bradley's SAT-based IC3 [15] and word-level algorithms based on SMT (Satisfiability Modulo Theories) are among the top contenders.

Unfortunately, there are still a sufficient number of cases where each of a variety of SAT- and BDD-based algorithms does best, that it has not been feasible to rely only on one. Therefore, all of these are supported. Until recently, the best result came from running several of these algorithms in parallel and killing the runs as soon as the first finished, exploiting the nearly ubiquitous multi-core environments found today. Experience has shown a super-linear speed-up using this method when checking many properties, over running only the overall single best algorithm. More recently, sophisticated combinations utilizing these and other "helper" algorithms in parallel [16] have come into play and now dominate.

Superimposed on top of the basic model-checking algorithms are a variety of proof strategies. Many of these are based on abstraction, the most important lever for lifting large state spaces. One is localization wherein portions of the design that are irrelevant to checking a given property are eliminated through abstraction. In its original formulation [54, 56] the algorithm iterates over abstractions determined by counterexamples on successive refinements. Ed Clarke et al. refined this algorithm with a SAT-based decision procedure [25]. In a significant SAT-based improvement [65], the successive abstractions are determined not by the counterexamples but by the unSAT clauses used to refute falsification of the property in the original model at a given depth. This "Abstraction-Refinement" loop has led to many further improvements of this basic idea, driven by the strength of the SAT solver in finding efficient refutations. Ranjit Jhala and Ken McMillan have extended the method using interpolation [49]. Through the successive improvements, the SAT solver is brought into play more and more as a deductive reasoning engine. Quantifier support is already in the works. One may speculate whether automated theorem proving will re-emerge through this thrust as truly automated deduction inspired by DPLL-style deductive procedures.

Other proof strategies in commercial play today include word-level algorithms that utilize SMT solvers, predicate abstraction, induction, abduction, symmetry reduction and (automatic) assume-guarantee reasoning. Assume-guarantee reasoning can be flat or can follow the design hierarchy.

The same infrastructure used for model checking is used for synthesis optimizations and automatic test bench synthesis for simulation. For the latter, BDDs or SAT are used for constraint-solving in the context of guided-random search. Alternatively, properties viewed as constraints are converted to generators that generate only legal inputs. All of these are in mainstream product use today [86].

For falsification, various restrictions are used. The depth of search can be restricted with bounded model checking. The input space can be restricted as well, especially by restricting the range of data-path inputs.

Symbolic trajectory evaluation [75] has seen extensive use in hardware verification, especially at Intel. It is a very specialized (restricted) form of symbolic model

checking developed by Randy Bryant and his student Carl Seger, in which properties are expressed exclusively in terms of the temporal logic *next-time* operator and Boolean connectives.

23.7 Future

The easiest prediction for the future of commercial computer-aided verification is that today's technology will catch up with that future. This means that commercial model checking will arrive for general software development—perhaps after a means has been found to provide the same sort of optimizations discussed above that synthesis routines provide for hardware, and a standardized functional property specification language is adopted. However, general software presents some daunting challenges, including pointer analysis and memory allocation. Today, most successful software model-checking methodologies circumvent these by focusing on the software control flow. On the other hand, software also offers some opportunities absent in hardware: much of software is inherently word-level, supporting a more sophisticated mathematical analysis unhindered by those pesky hardware bits.

For hardware (first) we are headed strongly to the support of hierarchical verification for top-down design. Languages such as SystemC and SystemVerilog are first attempts in this direction, but they do not address how to relate successive levels of abstraction. We need to be able to write a high-level design, verify it and then refine it successively to a low-level implementation target in a manner that guarantees that each successive refinement is consistent with its abstraction. In this way, properties verified at one level of abstraction are guaranteed to hold at all successive levels of refinement (and thus do not need to be reverified). This is a divide-and-conquer method that (potentially) supports the verification of global properties largely beyond the reach of today's commercial model checkers. Global properties are verified once and for all in high-level abstractions, and then are guaranteed to hold in all refinements. A promising way to construct refinements is in a manner that guarantees that it is correct by construction. This saves the overhead of additional verification, but places restrictions on the structure of refinements. However, refinements must be guided by the low-level architecture in any case, so this restriction may be acceptable. In effect it leads to a dual top-down/bottom-up design methodology [57]. With hierarchical verification the emphasis shifts from falsification to verification, as the correctness of the implementation depends upon verifying the correctness of the high-level abstractions.

The next easy prediction—because it is already happening—is integration of model checking with simulation for "hybrid" algorithms (cf. above) and utilization of model checking in sequential equivalence checking. The latter is natural because in the worst case (when states and nets cannot be mapped) sequential equivalence checking *is* model checking. Sequential equivalence checking may also be used to check the correctness of refinements, if they are not already correct by construction.

Such hybrid algorithms are also used to augment simulation test: a model checker can be used in a variety of ways as a diagnostic tool to help increase simulation coverage, by "guiding" simulation test vectors to critical parts of the design.

A problem not yet fully solved is how to "give credit" to model checking in the context of simulation coverage metrics, for properties already verified by the model checker.

With hierarchical verification comes the integration of model checking with design: since a high-level model will be verified before refinements are even designed, model checking becomes a design tool in which the correctness of architectures and algorithms are checked. In the course of checking algorithms, theorem proving may be brought into play, and in the future a theorem prover may be an accessory to every model checker. (Model checkers are already accessories to many theorem provers/proof checkers.) While this requires expertise, by this time a new generation of designers hopefully will be ready for the challenge. Note that, as an accessory, theorem proving can add its power without delaying the design process.

Theorem proving may re-emerge as a fully automatic DPLL-style deductive engine. There already is a trend in this direction (see above).

Finally, there will undoubtedly be ever greater use of pre-verified components ("IP"). This "reuse", a very important productivity enhancer, can fit in with hierarchical verification and top-down design. Moreover, when designs are out-sourced, a more reliable contract than one that describes the design through discourse is one that specifies the design formally, requiring contractually that the completed design will be verified for a specified list of properties.

23.8 Conclusion

Computer-aided verification technology has finally—it took 20 years—been transferred to the EDA industry in the form of equivalence checking, model checking, and constraint solving for guided-random simulation. Equivalence checking and, later, constraint solving were fairly easy to transfer because they were not subject to the three great impediments to the transfer of model checking: a required methodology change, the vicious circle of funding, and inventing an acceptable infrastructure for defining properties to be checked. The required methodology change for model checking was accomplished through a succession of small steps, each of which was small enough to avoid significant disruption of existing methodologies, while significant enough to demonstrate value. This is why the technology transfer took so long (and it still continues). The required infrastructure for defining functional properties was established through standardization of property specification languages. The gating issues for these technologies remains speed and capacity. Speed is continually improving as technology advances. Capacity is improving as well, but is limited by the intrinsic computational complexity of the technologies. To circumvent the capacity limitation, the only known general strategy is abstraction-based divide-and-conquer. This is where much of the cutting edge research is focused.

Acknowledgements Thanks to Ken McMillan and Moshe Vardi for illuminating discussions and to the anonymous reviewers for their many very useful remarks.

References

1. http://www.absint.de/astree/ accessed 20 Oct 2013
2. Accellera Property Specification Language Reference Manual (version 1.01). http://www.eda.org/vfv/docs/psl_lrm-1.01.pdf accessed 20 Oct 2013
3. Ahrens, F.: Why It's So Hard For Toyota To Find Out What's Wrong. The Washington Post p. G01 (2010). see http://www.washingtonpost.com/wp-dyn/content/article/2010/03/06/AR2010030602448_pf.html accessed 20 Oct 2013
4. Akers, S.B.: Binary decision diagrams. IEEE Trans. Comput. **C-27**(6), 509–516 (1978)
5. Alur, R.: Model checking: from tools to theory. In: Grumberg, O., Veith, H. (eds.) 25 Years of Model Checking. LNCS, vol. 5000, pp. 89–106. Springer, Heidelberg (2008)
6. Alur, R., Itai, A., Kurshan, R.P., Yannakakis, M.: Timing verification by successive approximation. In: von Bochmann, G., Probst, D. (eds.) Proc. CAV'92. LNCS, vol. 663, pp. 137–150. Springer, Heidelberg (1993). Also Inf. Comput. **118**(1), 142–157 (1995)
7. http://www.astree.ens.fr/ accessed 20 Oct 2013
8. Ball, T., Rajamani, S.K.: The SLAM project: debugging system software via static analysis. In: Launchbury, J., Mitchell, J.C. (eds.) Symposium on Principles of Programming Languages (POPL), pp. 1–3. ACM, New York (2002)
9. Barr, M.: Unintended acceleration and other embedded software bugs (2011). http://embeddedgurus.com/barr-code/2011/03/unintended-acceleration-and-other-embedded-software-bugs/ accessed 20 Oct 2013
10. Berry, G., Gonthier, G.: The ESTEREL synchronous programming language: design, semantics, implementation. Sci. Comput. Program. **19**(2), 87–152 (1992)
11. Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S., Engler, D.: A few billion lines of code later: using static analysis to find bugs in the real world. Commun. ACM **53**, 66–75 (2010)
12. Biere, A., Cimatti, A., Clarke, E., Fujita, M., Zhu, Y.: Symbolic model checking using SAT procedures instead of BDDs. In: Proc. 36th Design Automation Conference, pp. 317–320. IEEE, Piscataway (1999)
13. Blau, J.: AT&T's frame relay crash highlights vulnerable nature of data networks. Total Telcom (1998). http://www.totaltele.com/view.aspx?C=0&ID=433385 accessed 20 Oct 2013
14. Bledsoe, W.W., Loveland, D.W. (eds.): Automated Theorem Proving: After 25 Years, vol. 29. AMS, Providence (1984). Especially the paper "Proof-Checking, Theorem-Proving and Program Verification" by R.S. Boyer and J.S. Moore, 119–132
15. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) Proc. VMCAI 2011. LNCS, vol. 6538, pp. 70–87. Springer, Heidelberg (2011)
16. Brayton, R., Mishchenko, A.: ABC: an academic industrial-strength verification tool. In: Tonilli, T., Cook, B., Jackson, P. (eds.) Proc. CAV'10. LNCS, vol. 6174, pp. 24–40. Springer, Heidelberg (2010)
17. Bryant, R.: Computer-aided verification prize awarded. Not. Am. Math. Soc. **56**(11), 1456–1457 (2009)
18. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Trans. Comput. **35**(8), 677–691 (1986)
19. Bryant, R.E., Kukula, J.H.: Formal methods for functional verification. In: Kuehlmann, A. (ed.) The Best of ICCAD: 20 Years of Excellence in Computer-Aided Design, pp. 3–16. Kluwer Academic, Norwell (2003)
20. Büchi, J.R.: On a decision method in restricted second-order arithmetic. In: Nagel, E., Suppes, P., Tarski, A. (eds.) Methodology and Philosophy of Science, Proc., 1960 Stanford Intern. Congr., pp. 1–11. Stanford University Press, Stanford (1962)

21. Church, A.: Application of recursive arithmetics to the problem of circuit synthesis. In: Summaries of Talks Presented at the Summer Institute for Symbolic Logic, pp. 3–50 (1957). Communications Research Division, Institute for Defense Analysis
22. Clarke, E.M.: The birth of model checking. In: Grumberg, O., Veith, H. (eds.) 25 Years of Model Checking. LNCS, vol. 5000, pp. 1–26. Springer, Heidelberg (2008)
23. Clarke, E.M., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Form. Methods Syst. Des. **19**(1), 7–34 (2001)
24. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons for branching time temporal logic. In: Kozen, D. (ed.) Proc. Logic of Programs Workshop. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1982)
25. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM **50**(5), 752–794 (2003). A preliminary version was published as "Counterexample-Guided Abstraction Refinement". In: Emerson, E.A., Prasad, A. (eds.) Proc. CAV'00. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000)
26. Clarke, E.M., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004). LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)
27. NASA's metric confusion caused Mars orbiter loss (1999). http://www.cnn.com/TECH/space/9909/30/mars.metric/; see also http://www.thinkreliability.com/CM-MarsCO.aspx; both accessed 20 Oct 2013
28. Coe, T.: Inside the Pentium FDIV bug. Dr. Dobb's J. **20**, 129–135 (1995)
29. Cousot, P., Cousot, R.: Basic concepts of abstract interpretation. In: Jacquard, R. (ed.) Building the Information Society, pp. 359–366. Kluwer Academic, Norwell (2004)
30. http://www.coverity.com/products/static-analysis.html accessed 20 Oct 2013
31. Dill, D.L., Drexler, A.J., Hu, A.J., Yang, C.H.: Protocol verification as a hardware design aid. In: IEEE International Conference on Computer Design: VLSI in Computers and Processors, pp. 522–525. IEEE, Piscataway (1992)
32. Emanuelsson, P., Nilsson, U.: A comparative study of industrial static analysis tools. Electron. Notes Theor. Comput. Sci. **217**, 5–21 (2008). doi:10.1016/j.entcs.2008.06.039 accessed 20 Oct 2013
33. Emerson, E.A.: The beginning of model checking: a personal perspective. In: Grumberg, O., Veith, H. (eds.) 25 Years of Model Checking. LNCS, vol. 5000, pp. 27–45. Springer, Heidelberg (2008)
34. Emerson, E.A., Lei, C.L.: Efficient model checking in fragments of the propositional μ-calculus. In: Proc. LICS, pp. 267–278. IEEE, Piscataway (1986)
35. Fiskio-Lasseter, J., Sabry, A.: Putting operational techniques to the test: a syntactic theory for behavioral Verilog. Electron. Notes Theor. Comput. Sci. **26**, 34–51 (1999)
36. Floyd, R.W.: Assigning meanings to programs. In: Mathematical Aspects of Computer Science, vol. 19, pp. 19–32 (1967)
37. Lucent's Bell introduces FormalCheck. Electronic News (1997)
38. Foster, H.: Prologue: the 2010 Wilson research group functional verification study (2011). http://blogs.mentor.com/verificationhorizons/blog/2011/03/30/prologue-the-2010-wilson-research-group-functional-verification-study/ accessed 20 Oct 2013
39. Foster, H.D., Krolnik, A.C., Lacey, D.J.: Assertion Based Design, 2nd edn. Kluwer Academic, Norwell (2004)
40. Goldstine, H.H., von Neumann, J.: Planning and Coding of the Problems for an Electronic Computing Instrument. Institute for Advanced Study, Princeton (1947)
41. Gordon, M., Milner, R., Wadsworth, C.P.: Edinburgh LCF: A Mechanised Logic of Computation. LNCS, vol. 78 (1979)
42. http://www.grammatech.com/products/codesonar/smashproof_analysis.html accessed 20 Oct 2013
43. Grumberg, O.: McMillan receives CAV award. Not. Am. Math. Soc. **57**(10), 1318 (2010)
44. Grumberg, O., Veith, H. (eds.): 25 Years of Model Checking. LNCS, vol. 5000. Springer, Heidelberg (2008)

45. Gupta, A.: Alur and Dill receive computer-aided verification award. Not. Am. Math. Soc. **55**(11), 1429 (2008)
46. Hoare, C.A.R.: An axiomatic basis for programming. Commun. ACM **12**(10), 576–580 (1969)
47. Huuck, R., Lukoschus, B., Frehse, G., Engell, S.: Compositional verification of continuous-discrete systems. In: Engell, S., Frehse, G., Schnieder, E. (eds.) Modelling, Analysis, and Design of Hybrid Systems. LNCS, vol. 279, pp. 225–244. Springer, Heidelberg (2002)
48. Toyota shareholders in US sue over fallen stock price. JapanToday (2010). http://www.japantoday.com/category/business/view/toyota-shareholders-in-us-sue-over-fallen-stock-price accessed 06 Aug 2013
49. Jhala, R., McMillan, K.L.: Interpolant-based transition relation approximation. In: Etessami, K., Rajamani, S. (eds.) Proc. CAV'05. LNCS, vol. 3576, pp. 39–51. Springer, Heidelberg (2005)
50. Kaufmann, M., Moore, J.S.: Some key research problems in automated theorem proving for hardware and software verification. Rev. R. Acad. Cienc. Exactas Fís. Nat., Ser. a Mat. **98**(1), 185–195 (2004)
51. Khasidashvili, Z., Gavrielov, G., Melham, T.: Assume-guarantee validation for STE properties within an SVA environment. In: Biere, A., Pixley, C. (eds.) Proc. FMCAD'09, pp. 108–115. IEEE, Piscataway (2009)
52. http://docs.klocwork.com/Insight-9.6/Klocwork_Truepath accessed 15 Sep 2015
53. Kozen, D.: Results on propositional μ-calculus. Theor. Comput. Sci. **27**, 333–354 (1983)
54. Kurshan, R.P.: Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach. Princeton University Press, Princeton (1994)
55. Kurshan, R.P.: Formal verification in a commercial setting. In: Proc. DAC'97, vol. 34, pp. 258–262 (1997)
56. Kurshan, R.P.: Program verification. Not. Am. Math. Soc. **47**(5), 534–545 (2000)
57. Kurshan, R.P.: Scaling commercial verification to larger systems. In: Yorav, K. (ed.) Hardware and Software: Verification and Testing. LNCS, vol. 4899, pp. 8–13. Springer, Heidelberg (2008)
58. Kurshan, R.P.: Verification technology transfer. In: Grumberg, O., Veith, H. (eds.) 25 Years of Model Checking. LNCS, vol. 5000, pp. 46–64. Springer, Heidelberg (2008)
59. Lee, C.Y.: Representations of switching circuits by binary-decision programs. Bell Syst. Tech. J. **38**, 985–999 (1959)
60. Leveson, N.G., Turner, C.S.: An investigation of the Therac-25 accidents. IEEE Comput. **26**(7), 18–41 (1993)
61. Lions, J.L.: Ariane 5 flight 501 failure (1996). http://web.archive.org/web/20000815230639/www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html accessed 20 Oct 2013
62. Lohr, S.: AT&T data network fails and commerce takes a hit. The New York Times (1998). http://www.nytimes.com/1998/04/15/business/at-t-data-network-fails-and-commerce-takes-a-hit.html accessed 20 Oct 2013
63. McMillan, K.L.: Symbolic Model Checking. Kluwer Academic, Norwell (1993)
64. McMillan, K.L.: Interpolation and SAT-based model checking. In: Hunt, W.A. Jr., Somenzi, F. (eds.) Proc. CAV'03. LNCS, vol. 2725, pp. 1–13. Springer, Heidelberg (2003)
65. McMillan, K.L., Amla, N.: Automatic abstraction without counterexamples. In: Garavel, H., Hatcliff, J. (eds.) Proc. TACAS'03. LNCS, vol. 2619, pp. 2–17. Springer, Heidelberg (2003)
66. Mead, C., Conway, L.: Introduction to VLSI Systems. Addison-Wesley, Reading (1979)
67. Naur, P.: Proof of algorithms by general snapshots. BIT **6**(4), 310–316 (1966)
68. http://www.parasoft.com/jsp/technologies/code_analysis.jsp accessed 20 Oct 2013
69. Pnueli, A.: The temporal logic of programs. In: Proc. Eighteenth FOCS, pp. 46–57. IEEE, Piscataway (1977)
70. http://www.mathworks.com/products/polyspace/ accessed 20 Oct 2013
71. Queille, J.P., Sifakis, J.: Specification and verification of concurrent systems in CESAR. In: Dezani-Ciancaglini, M., Montanari, U. (eds.) Proc. Intl. Symp. on Programming. LNCS, vol. 137, pp. 337–351. Springer, Heidelberg (1982)

72. Rabin, M.O., Scott, D.: Finite automata and their decision problems. IBM J. Res. Dev. **3**, 114–125 (1959)
73. Robinson, J.A.: Machine-oriented logic based on the resolution principle. J. ACM **12**, 23–41 (1965)
74. Rudin, H., West, C.: A validation technique for tightly coupled protocols. IEEE Trans. Comput. **31**(7), 630–636 (1982)
75. Seger, C.J.H., Bryant, R.E.: Formal verification by symbolic evaluation of partially-ordered trajectories. Form. Methods Syst. Des. **6**(2), 147–190 (1995)
76. Shannon, C.E.: The synthesis of two-terminal switching circuits. Bell Syst. Tech. J. **28**, 59–98 (1949)
77. Sterling, B.: PART ONE: crashing the system (1990). http://www.farcaster.com/sterling/part1.htm accessed 15 Sep 2015
78. Travis, P.: Why the AT&T network crashed. Telephony **218**(4), 11 (1990). See also http://www.phworld.org/history/attcrash.htm and http://tech-insider.org/data-security/research/1990/0126.html, both accessed 15 Sep 2015
79. Turing, A.: On computable numbers, with an application to the Entscheidungsproblem. Proc. Lond. Math. Soc. **42** (1936)
80. Vardi, M.: Ball and Rajamani receive 2011 CAV award. Not. Am. Math. Soc. **58**(11), 1597 (2011)
81. Vardi, M.Y.: From Church and Prior to PSL. In: Grumberg, O., Veith, H. (eds.) 25 Years of Model Checking. LNCS, vol. 5000, pp. 150–171. Springer, Heidelberg (2008)
82. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification. In: Proc. (1st) IEEE Symposium on Logic in Computer Science (LICS), pp. 322–331 (1986)
83. Waxer, C.: The hidden cost of IT security. Network Security Journal (2006). http://www.networksecurityjournal.com/features/hidden-cost-of-IT-security-041607/ accessed 20 Oct 2013
84. West, C.: Generalized technique for communication protocol validation. IBM J. Res. Dev. **22**, 393–404 (1978)
85. Whitehead, A.N., Russell, B.: Principia Mathematica. Cambridge University Press, Cambridge (1910–1913)
86. Yuan, J., Pixley, C., Aziz, A.: Constraint-Based Verification. Springer, Heidelberg (2006)

Chapter 24
Functional Specification of Hardware via Temporal Logic

Cindy Eisner and Dana Fisman

Abstract In the late 1970s, Amir Pnueli suggested that functional properties of reactive systems be formally expressed in temporal logic. For model checking such a logic to be possible, it must have sufficient expressive power, its semantics must be formally defined in a rigorous way, and the complexity of model checking it must be well understood and reasonable. In order to allow widespread adoption in industry, there is an additional requirement: functional specification must be made easy, allowing common properties to be expressed intuitively and succinctly. But while adding syntax is simple, defining semantics without breaking properties of the existing semantics is a different story. This chapter is about the various extensions to temporal logic included in the IEEE standards PSL and SVA, their motivation, and the subtle semantic issues encountered in their definition.

24.1 Introduction

In his seminal 1977 paper [62], Amir Pnueli first suggested that functional properties of reactive systems be formally expressed in temporal logic (Chap. 2). While the proposal was widely accepted in academia, the exact nature of the temporal logic was a topic of long debate. In particular, the nature of time was widely discussed—whether it should be linear, where each point has a unique next future, or branching, where each point may have multiple next futures. The choice has implications for expressivity as well as the complexity of model checking—see [69] for a survey. Another focus of debate surrounded the question of how to augment LTL (Linear Temporal Logic), which has the expressive power of star-free ω-regular languages, to a temporal logic that has the power of full ω-regular languages. Proposals included automata connectives, second-order quantification, and grammar operators—see [70] for a full exposition.

C. Eisner
IBM Research – Haifa, Haifa, Israel

D. Fisman (✉)
Ben-Gurion University, Be'er Sheva, Israel
e-mail: dana@cs.bgu.ac.il

E.M. Clarke et al. (eds.), *Handbook of Model Checking*,
DOI 10.1007/978-3-319-10575-8_24

With the invention of symbolic model checking in the early 1990s [57], the industrial applicability of model checking began to be recognized, and in 1998 the Accellera Formal Verification Technical Committee began an effort to standardize a temporal logic for use in the hardware industry. Requirements were gathered from industrial users of hardware model checking as to what kinds of things they needed to be able to express more easily than they could in existing temporal logics. This effort eventually resulted in two leading IEEE standards, PSL (Property Specification Language, IEEE Standard 1850 [43, 44]) and SVA (SystemVerilog Assertions, IEEE Standard 1800 [45, 46]).

Both PSL and SVA are based on the linear paradigm, and in particular on LTL [62], while PSL includes as well an optional branching extension based on CTL [17] (see also Chap. 2 for a discussion of LTL and CTL). The approach of both to extending the expressive power of LTL to that of ω-regular languages is to add regular expressions and the *suffix implication* operator, which gives implication a temporal flavor by making the consequent dependent on the end of the regular expression used as an antecedent. They both also include a number of specialized operators to allow natural specification of sampling abstractions (e.g., hardware clocks) and truncated paths (e.g., hardware resets). Finally, both provide local variables, which can be seen as a mechanism for both declaring quantified variables and constraining their behavior. Local variables are another way to extend the expressive power of LTL, and also provide much succinctness.

When extending a temporal logic with a new operator, it is important not to break properties of the existing semantics or those of the extension. For instance, if a common property such as distributivity of union over intersection is broken, two formulas that are intuitively equivalent may no longer be so, and, even worse, a tool relying on this property may produce an incorrect result. Even if we are willing to review and fix existing algorithms, breaking properties of the existing semantics or its extensions might compromise a user's intuition, making the logic effectively unusable.

Extending a logic without breaking existing properties of the semantics is not trivial, and many early attempts failed. For example, an early attempt at defining the semantics of a clock operator broke the fixed point characterization of LTL's strong until operator in terms of the next operator, and an early attempt at defining local variables broke the distributivity of union over intersection.

In this chapter we will examine the major issues raised by the extension of LTL in order to meet the needs of industry. Since syntax is not the interesting part of the story, we will use a mathematical syntax that allows us to illustrate the semantic issues addressed by both standards without emphasizing either one. In each section we present only a fragment of the semantics relevant to the discussion at hand. The full syntax and formal semantics of PSL and SVA can be found in the corresponding standards [43–46]. Introductions to PSL and SVA aimed at users can be found in [19] and [16], respectively.

The current standards are not necessarily the final word; standards keep evolving to meet the growing and changing needs of the industry. In Sect. 24.7 we touch briefly on some directions for future research.

Let P be a set of atomic propositions and b a Boolean expression (propositional formula) over P. The syntax of LTL is given by:

$$f ::= b \mid \neg f \mid f \wedge f \mid f \vee f \mid \mathrm{X}f \mid \mathrm{G}f \mid \mathrm{F}f \mid [f\ \mathrm{U}\ f] \mid [f\ \mathrm{W}\ f]$$

Let f, f_1, f_2 be LTL formulas over P, and w an infinite word over $\Sigma = 2^P$. We use $\ell \Vdash b$ to denote that a letter $\ell \in \Sigma$ satisfies a given Boolean expression b. The semantics of LTL is given by:

- $w \models b \iff w(0) \Vdash b$
- $w \models \neg f \iff w \not\models f$
- $w \models f_1 \wedge f_2 \iff w \models f_1$ and $w \models f_2$
- $w \models \mathrm{X}\ f \iff w(1..) \models f$
- $w \models [f_1\ \mathrm{U}\ f_2] \iff \exists k$ such that $w(k..) \models f_2$ and $\forall j < k$ we have $w(j..) \models f_1$

$f_1 \vee f_2 \equiv \neg(\neg f_1 \wedge \neg f_2) \quad \mathrm{F}f \equiv [\mathrm{TRUE}\ \mathrm{U}\ f] \quad \mathrm{G}f \equiv \neg\mathrm{F}\neg f \quad [f_1\ \mathrm{W}\ f_2] \equiv [f_1\ \mathrm{U}\ f_2] \vee \mathrm{G}f_1$

Fig. 1 The semantics of LTL [62]

24.2 From LTL to Regular-Expression-Based Temporal Logic

The temporal logics PSL and SVA are *linear* temporal logics. In contrast to branching time logics, they follow the view that every point in time has a unique future, and more precisely a unique next time point (see Chap. 2 for more on linear and branching time logics). At the core of PSL and SVA lies the temporal logic LTL standing for *linear time logic*, as proposed by Pnueli [62].

Formulas of LTL are phrased with respect to a set of atomic propositions P, and are interpreted with respect to an infinite word over the alphabet $\Sigma = 2^P$, whose letters represent points in time. We use the notation $[\![\varphi]\!]$ to denote the set of words on which formula φ holds. We number the letters of word w starting from 0, like this: $w = \ell_0\ell_1\ell_2\cdots$, and we use $w(k)$ to denote the $k+1^{st}$ letter of w (since counting starts at 0). We use $w(j..k)$ to denote the finite subword of w starting at $w(j)$ and ending at $w(k)$. We use $w(j..)$ to denote the suffix of w starting at $w(j)$. In addition to the usual propositional operators $\neg$, $\wedge$, and $\vee$, there are temporal operators X (read "next"), G (read "globally"), F (read "eventually"), U (read "strong until", or simply "until") and W (read "weak until") as shown in Fig. 1. If φ is a formula of LTL, the formula $\mathrm{X}\varphi$ holds if φ holds at the next point in time. The formula $\mathrm{G}\varphi$ holds if φ holds now and at every point in the future. The formula $\mathrm{F}\varphi$ holds if φ holds now or at some point in the future. The formula $[\varphi\ \mathrm{U}\ \psi]$ holds if ψ holds now or at some point in the future, and in addition φ holds at every point in time until ψ holds. The formula $[\varphi\ \mathrm{W}\ \psi]$ holds if either $[\varphi\ \mathrm{U}\ \psi]$ holds, or φ holds forever. The operators G, F, and W can be derived from the other operators as shown in Fig. 1.

Using LTL we can express many interesting properties. For instance the property $\mathrm{G}(send \rightarrow \mathrm{XX}\ received)$ states that signal $received$ should be asserted exactly two time points after signal $send$ is asserted. The property $\mathrm{G}(send \rightarrow [busy\ \mathrm{U}\ received])$ states that signal $received$ should be asserted some time after signal $send$ is asserted, and in the meantime signal $busy$ should be asserted.

One of the most important properties of a formalism is its *expressive power*. Besides LTL, other known formalisms for reasoning on infinite words are first-order logic over the naturals, monadic second-order logic of one successor, automata on infinite words, and ω-regular expressions. In terms of expressive power we can roughly classify these into two sets: those recognizing all the ω-regular languages, and those recognizing a strict subset of those, referred to as the *star-free ω-regular* languages.

It follows from a sequence of results of [31, 47, 68] that LTL belongs to the latter (see also [27]). For example, as shown by Wolper [74], the property "p holds on every even position" is not expressible in LTL. Note that simply adding regular expressions, without the ω operator, will not achieve the expressive power of ω-regular languages. However, we can achieve that expressive power through the addition of regular expressions plus the suffix implication operator, as discussed in Sect. 24.2.1.

Before continuing, we note that in regular expressions as used in PSL and SVA, the syntactic atoms are Boolean expressions, more than one of which may hold on a given letter of the alphabet. This is in contrast to standard regular expressions, in which the syntactic atoms are mutually exclusive. As in standard regular expressions, we use the notation $\mathcal{L}(r)$ to denote the set of words recognized by the regular expression.

24.2.1 Adding Expressive Power—Suffix Implication

Suffix implication, denoted $\mapsto$, and also known as *triggers*, is an operator taking two arguments, a regular expression r and a temporal formula φ. Suffix implication gives implication a temporal flavor by making the consequent dependent on the end of the regular expression used as an antecedent. Intuitively, $r \mapsto \varphi$ holds on a word w if for every prefix of w recognized by r, the suffix of w, starting at the letter on which that prefix ends, satisfies φ.

The suffix implication operator, first proposed in the context of temporal logic in [8], is reminiscent of the modality $[\alpha]\varphi$ of dynamic logic [28, 38, 63]. It is different than the other attempts at combining temporal and dynamic logic ([35, 37, 39, 71]) in that it borrows the dynamic modalities but remains state-based as in temporal logic rather than action-based as in dynamic logic.

Augmenting LTL with the suffix implication operator $\mapsto$, the semantics of which are shown in Fig. 2, increases the expressive power to that of ω-regular languages, as stated in the following theorem.

Theorem 1 ([6, 51]) *Let L be an ω-regular language. Then there exists a formula φ of LTL extended with suffix implication such that $[\![\varphi]\!] = L$.*

Using suffix implication, we can express the property that p holds on every even position as follows:

$$\big((\text{TRUE}, \text{TRUE})^*\big) \mapsto p \tag{1}$$

Let r be a regular expression, φ a formula of LTL extended with suffix implication and its dual, and w an infinite word over $\Sigma = 2^P$.

- $w \models r \ \mapsto \ \varphi \iff \forall j$ if $\ w(0..j) \in \mathcal{L}(r)$ then $w(j..) \models \varphi$
- $w \models r \ \diamond\!\!\rightarrow \varphi \iff \exists j$ s.t. $w(0..j) \in \mathcal{L}(r)$ and $\ w(j..) \models \varphi$
- All other LTL operators are as usual.

Fig. 2 Semantics of LTL extended with suffix implication [6–8, 43–46] and its dual

Although suffix implication extends the expressive power to that of ω-regular languages, expressive power is not the only issue. In order to make suffix implication really useful to users, we must provide syntactic support for commonly needed patterns. These are the subjects of Sect. 24.2.2 and Sect. 24.2.3.

24.2.2 Adding Succinctness

24.2.2.1 Counting

The User's Point of View Consider the property expressing that if a request is acknowledged (signal *ack* is asserted four to six cycles after *req* is asserted, and in the meantime signal *busy* is asserted), then signal *busy* should remain asserted until *done* holds. This property can be expressed as follows in LTL:

$$\begin{aligned} \mathrm{G}\,\big(req \rightarrow \mathrm{X}\big(busy \rightarrow \mathrm{X}\big(busy \rightarrow \mathrm{X}\big(busy \rightarrow \mathrm{X}\big(\big(ack \rightarrow [busy\ \mathrm{U}\ done]\big) \\ \wedge \big(busy \rightarrow \mathrm{X}\big(\big(ack \rightarrow [busy\ \mathrm{U}\ done]\big) \\ \wedge \big(busy \rightarrow \mathrm{X}\big(ack \rightarrow [busy\ \mathrm{U}\ done]\big)\big)\big)\big)\big)\big)\big)\big) \end{aligned} \tag{2}$$

However, increase "four to six cycles" and the property soon becomes unwieldy. Thus the user would like there to be an easier way.

Bare suffix implication is not much help. We could use three separate formulas, or we could use the regular expression union operator $\cup$ (distinguished from U, denoting the strong until operator) to write:

$$\begin{aligned} \mathrm{G}\,\big(\big(req \cdot \big(\ (busy \cdot busy \cdot busy)\ \cup\ (busy \cdot busy \cdot busy \cdot busy) \cup \\ (busy \cdot busy \cdot busy \cdot busy \cdot busy)\big) \cdot ack\big) \mapsto [busy\ \mathrm{U}\ done]\big) \end{aligned} \tag{3}$$

However, add the ability to count and Formulas (2) and (3) can be expressed much more clearly. Let $\cdot$ and * denote concatenation and Kleene star, respectively, let the repetition operator $r[^*k]$ abbreviate r concatenated to itself k times, and let $r[^*i..j]$ abbreviate $\bigcup_{k=i}^{j} r[^*k]$. Then we can write simply:

$$\mathrm{G}\,\big(\big(req \cdot busy\big[^*3..5\big] \cdot ack\big) \mapsto [busy\ \mathrm{U}\ done]\big) \tag{4}$$

The user would like counting operators, so that Formula (2) can be expressed simply as Formula (4). Other desirable counting operators are the goto repetition operator $b[\rightarrow k]$, abbreviating $(\neg b^* \cdot b)[^*k]$ and discussed in Sect. 24.2.2.2, and the non-consecutive repetition operator, $b[= k]$, abbreviating $(\neg b^* \cdot b)[^*k] \cdot \neg b^*$.

Semantic Issues Counting operators are simple syntactic sugar, and as such do not add expressive power. However, they do affect succinctness.

Theorem 2 ([32, 49]) *Regular expressions with counting operators are doubly exponential more succinct than DFAs and exponentially more succinct than standard regular expressions and NFAs.*

24.2.2.2 First Match

The User's Point of View Consider the properties "the first occurrence of *ack* after every *req* is followed by *gnt*". In LTL this would be

$$\mathrm{G}\left(req \rightarrow \mathrm{X}\left[\neg ack \ \mathrm{U}\ (ack \wedge \mathrm{X} gnt)\right]\right) \tag{5}$$

The user would like a more direct way to express this. Using the *goto* operator, $b[\rightarrow]$, abbreviating $\neg b^* b$ and supported by both PSL and SVA, achieves this as follows:

$$\mathrm{G}\left(\left(req \cdot ack[\rightarrow]\right) \mapsto (gnt)\right) \tag{6}$$

Semantic Issues The goto operator is a kind of counting operator, and like the repetition operators of Sect. 24.2.2.1, designating the ith occurrence of a Boolean expression is achieved through syntactic sugaring: $b[\rightarrow i]$ abbreviates $(\neg b^* b)[^*i]$. SVA includes as well the *first match* operator, denoted here by $\mathrm{FM}(r)$, designating the first occurrence of a regular expression. First match adds succinctness, but does not add expressive power. On the other hand, while intuitive, its semantics is not easily derived from other operators, so it is more than syntactic sugar.

Formally, $\mathcal{L}(\mathrm{FM}(r)) = \{w \in \mathcal{L}(r) \mid w = uv \text{ and } u \in \mathcal{L}(r) \text{ implies that } v = \varepsilon\}$. The origin of first match is in [59], which also provides the *fail* operator, capturing the set of shortest words w such that w is not in $\mathcal{L}(r)$. Formally, $\mathcal{L}(\mathrm{FAIL}(r)) = \{w \notin \mathcal{L}(r) \mid w = uv \text{ and } u \notin \mathcal{L}(r) \text{ implies that } v = \varepsilon\}$. Intuitively, $\mathcal{L}(\mathrm{FM}(r))$ is the set of shortest words in $\mathcal{L}(r)$ ("good prefixes") and $\mathcal{L}(\mathrm{FAIL}(r))$ is the set of shortest words with no extension in $\mathcal{L}(r)$ ("bad prefixes"). These operators raise issues related to the distinction between bad prefixes and informative prefixes and the related complexity issues—see Sect. 24.4.3.

24.2.2.3 Intersection

The User's Point of View Consider the property asserting that if a print request is issued (*glbl_prnt* is asserted), then all three printers should issue either a success or

error message (assertion of $succ_i$ or err_i, or in short, assertion of se_i) before signal $prnt\_done$ is asserted. If we try to formulate it using only the standard operators, we see that we need to account for all the possible orders in which the printers will issue the corresponding message.

The user would like an easier way. Using the intersection operator and the non-consecutive repetition operator $b[=k] \equiv (\neg b^* \cdot b)[^*k] \cdot \neg b^*$, the property can be stated as follows:

$$\mathrm{G}\left(glbl\_prnt \rightarrow \left(\left(se_1[=1] \cap se_2[=1] \cap se_3[=1] \cap prnt\_done[=0]\right) \cdot prnt\_done\right)\right) \tag{7}$$

Semantic Issues The semantics of intersection is straightforward, and is given by $\mathcal{L}(r_1 \cap r_2) = \mathcal{L}(r_1) \cap \mathcal{L}(r_2)$. It is well known that regular sets are closed under intersection [42], thus there is no increase in expressive power. However, there is an increase in succinctness, as stated by the following theorem.

Theorem 3 ([32, 33]) *Regular expressions with intersection are doubly exponential more succinct than standard regular expressions and DFAs and exponentially more succinct than NFAs.*

24.2.2.4 Fusion

The User's Point of View Let a *send transaction* be a sequence of cycles where $processing$ holds for some number (possibly zero) of cycles, then $sending$ holds for some number (possibly zero) of cycles and then $sent$ holds, and consider the property that if a granted send-request (assertion of $send$ followed by assertion of gnt) is followed by a successful send transaction (that starts at the same cycle as the grant), then signal ok should be asserted at the same time that $sent$ is asserted.

Our formula must distinguish between three different ways in which a send transaction might begin, so we get something like this:

$$\mathrm{G}\left(\left(send \cdot \left((gnt \wedge sent) \cup \left((gnt \wedge sending) \cdot sending^* \cdot sent\right) \cup \left((gnt \wedge processing) \cdot processing^* \cdot sending^* \cdot sent\right)\right)\right) \mapsto (ok)\right) \tag{8}$$

The user would like a more straightforward way. Using the *fusion* operator, a kind of overlapping concatenation denoted $\circ$, we can formulate this more succinctly as follows:

$$\mathrm{G}\left(\left((send \cdot gnt) \circ \left(processing^* \cdot sending^* \cdot sent\right)\right) \mapsto (ok)\right) \tag{9}$$

Semantic Issues The fusion of languages U and V over Σ, denoted $U \circ V$, is the set $\{u\ell v \mid \ell \in \Sigma, u\ell \in U, \text{ and } \ell v \in V\}$. Like the counting operators and intersection, fusion does not add expressive power, but does add succinctness.

Theorem 4 ([41]) *Regular expressions with fusion are exponentially more succinct than standard regular expressions and DFAs.*

Henceforth we refer to regular expressions augmented with intersection and fusion as *semi-extended regular expressions (SEREs)* (note that the term *extended regular expression* is sometimes used for regular expressions with complementation).

24.2.2.5 Past Operators

The User's Point of View Consider the following property: "signal $dt\_complete$ should be asserted if and only if a data transfer has just completed," where a data transfer is an assertion of signal $data\_start$ followed by some number of assertions of $data$ followed by $data\_end$. One direction of the implication is easy:

$$\mathrm{G}\left(data\_start \cdot data^* \cdot data\_end\right) \mapsto dt\_complete \tag{10}$$

However, the other direction is extremely difficult to express without past operators, as it involves negation of a regular expression and thus determinization. The user would like to be able to say simply:

$$\mathrm{G}\left(dt\_complete \leftrightarrow ended\left(data\_start \cdot data^* \cdot data\_end\right)\right) \tag{11}$$

where, for SERE r, $ended(r)$ holds in position j of word w iff there exists $i \leq j$ such that $w(i..j) \in \mathcal{L}(r)$. Both PSL and SVA supply the $ended()$ operator, which can be used wherever a Boolean expression can be used.

Semantic Issues Adding past operators to LTL (for every LTL operator its dual past operator) does not increase the expressive power:

Theorem 5 ([54]) *LTL with (future and) past operators has the same expressive power as LTL (with just future operators).*

It is easy to see by automata construction that the analogous result holds for LTL augmented with suffix implication and the past operator $ended()$. Past operators do, however, add succinctness [52] (see also Chap. 2).

24.2.3 Distinguishing Between Weak and Strong Regular Expressions

The User's Point of View Recall Formula (4), repeated below as Formula (12):

$$\mathrm{G}\left(\left(req \cdot busy\left[^*3..5\right] \cdot ack\right) \mapsto [busy\ \mathrm{U}\ done]\right) \tag{12}$$

As long as regular expressions are available, many users prefer to avoid LTL completely. Can we use regular expressions to formulate the right-hand side of Formula (12) as well? The regular expression $(busy^* \cdot done)$ seems a good candidate to represent the right-hand side. But in LTL, it is possible to distinguish between the formula $[busy \text{ U } done]$, using the *strong* until operator, requiring $done$ to eventually hold, and the formula $[busy \text{ W } done]$, using the *weak* until operator, which holds also if $done$ never holds and $busy$ holds forever.

Thus the user would like a syntax distinguishing between *strong* and *weak* versions of a regular expression. Using $r!$ to denote a strong regular expression, Formula (4) can be rephrased as follows:

$$\text{G}\left(req \cdot busy\left[^*3..5\right] \cdot ack\right) \mapsto \left(busy^* \cdot done\right)! \tag{13}$$

Similarly, the weak version of Formula (12):

$$\text{G}\left(req \cdot busy\left[^*3..5\right] \cdot ack\right) \mapsto [busy \text{ W } done] \tag{14}$$

can be rephrased as follows, where r (vs. $r!$) denotes a weak regular expression:

$$\text{G}\left(req \cdot busy\left[^*3..5\right] \cdot ack\right) \mapsto \left(busy^* \cdot done\right) \tag{15}$$

Semantic Issues The intended semantics of a strong regular expression is clear. An infinite word satisfies a strong regular expression if it has a prefix in the language of the regular expression. For a weak regular expression, roughly speaking, we would like to say that we can get stuck in an infinite loop of a starred subexpression. Intuitively, this seems to require that every prefix of the observed word has some extension in the language of the regular expression.

While initially pleasing, this does not give the desired semantics. To see why, recall that we want the weak regular expression $(busy^* \cdot done)$ to be equivalent to $[busy \text{ W } done]$ and the weak regular expression $(busy^* \cdot \text{FALSE})$ to be equivalent to $[busy \text{ W FALSE}]$. However, using the definition that every prefix has some extension in the language of the regular expression works for $(busy^* \cdot done)$ but not for $(busy^* \cdot \text{FALSE})$, whose language is empty over the alphabet 2^P. Note that FALSE can be replaced with a complicated unsatisfiable formula, and so the situation is not necessarily easy to identify syntactically.

The $\top, \bot$ approach, proposed in [25], shown in Fig. 3 and adopted by PSL 1850–2005 [43] and SVA [45, 46], addresses this issue (see also Sect. 24.4.2). The idea is that $\top$ and $\bot$ are special letters, with the following properties: $\top$ satisfies any Boolean expression, including FALSE, and $\bot$ satisfies no Boolean expression, including TRUE. The inductive definition of the temporal logic makes the word $\top^\omega$ satisfy every temporal logic formula, whereas $\bot^\omega$ satisfies no temporal logic formula. This way, we can talk about extension without the need to worry about unsatisfiable eventualities of a regular expression.

Let r be a regular expression (but not a SERE, see below). Under the $\top, \bot$ semantics, $r!$ and r are related in the same way as other pairs of strong and weak LTL operators [22] . In analogy to the *safety* and *liveness* components of a formula [2, 3],

Let r be a regular expression, let $\top$ be a special letter such that $\top \not\models b$ for any Boolean expression b (including FALSE), and let w be an infinite word over $\Sigma = 2^P \cup \{\top, \bot\}$.

- $w \models r! \iff \exists j$ s.t. $w(0..j) \in \mathcal{L}(r)$
- $w \models r \iff \forall j\ w(0..j)\top^\omega \models r!$

Fig. 3 Semantics of strong and weak regular expressions [25] adopted by PSL 1850–2005 [43] and SVA [45, 46]

Let $\top$ and $\bot$ be special letters such that $\top \not\models b$ for any Boolean expression b (including FALSE), and $\bot \not\models b$ for any b (including TRUE). For some alphabet Γ, let Γ^∞ denote $\Gamma^* \cup \Gamma^\omega$. Let φ be a formula in LTL extended with weak and strong regular expressions. Then its weak and strong components, denoted $weak(\varphi)$ and $strong(\varphi)$ respectively, are defined as follows.

- $weak(\varphi) = \{w \in (\Sigma \cup \{\top, \bot\})^\infty \mid \forall \text{ finite } u \preceq w : u\top^\omega \in [\![\varphi]\!]\}$
- $strong(\varphi) = \{w \in (\Sigma \cup \{\top, \bot\})^\infty \mid \exists \text{ finite } u \preceq w : u\bot^\omega \in [\![\varphi]\!]\}$

Fig. 4 Weak and strong components [22]

[22] first defines the *weak* and *strong* component of a formula as the topological closure and interior, respectively, over the extended alphabet $\Sigma \cup \{\top, \bot\}$, and shows that such a definition is equivalent to the definition of Fig. 4. It then states the following characterization.

Theorem 6 ([22]) *Let φ be a formula of LTL extended with weak and strong regular expressions in positive normal form. Let φ_w be the formula obtained by weakening all operators (replacing $r!$ with r, X! with X and U with W). Let φ_s be the formula obtained by strengthening all operators. Then,*

$$\bullet\ [\![\varphi_w]\!] = weak(\varphi) \qquad \bullet\ [\![\varphi_s]\!] = strong(\varphi)$$

It is shown in [21, 22] that for the semantics of PSL 1850–2005 [43] and of SVA [45, 46], the characterization does not hold for SEREs, i.e., when intersection and fusion are included, because of the presence of *structural contradictions*. Structural contradictions are SEREs that are unsatisfiable (i.e., whose language is empty) due to their structure. That is, for any replacement of the propositions in a SERE, the language of the SERE remains empty. For example, $p \cap (p \cdot q)$ is a structural contradiction, while $(p \cdot q) \cap (p \cdot \neg q)$ is a contradiction, but not a structural one. A semantics fixing this problem was proposed in [21] and was adopted by PSL 1850–2010 [44]. It defines, in addition to the traditional $\mathcal{L}(r)$, the language of finite proper prefixes of a SERE, denoted $\mathcal{F}(r)$, and the loop language of a SERE, denoted $\mathcal{I}(r)$, shown in Fig. 5. Then the truncated semantics of strong and weak SEREs are as shown in Fig. 6. The topological characterization for this semantics appears in [23].

It is instructive to illustrate the difference between the weak/strong components and the safety/liveness components. Often they coincide, but not always. For exam-

Let ε be the empty word, let λ denote the empty regular expression, let b be a Boolean expression, let r, r_1 and r_2 be SEREs and let $\widehat{\Sigma} = \Sigma \cup \{\top, \bot\}$.

$\mathcal{L}(\lambda) = \{\varepsilon\}$	$\mathcal{L}(r_1 \cdot r_2) = \mathcal{L}(r_1) \cdot \mathcal{L}(r_2)$	$\mathcal{L}(r_1 \cup r_2) = \mathcal{L}(r_1) \cup \mathcal{L}(r_2)$
$\mathcal{L}(b) = \{\ell \in \widehat{\Sigma} \mid \ell \models b\}$	$\mathcal{L}(r_1 \circ r_2) = \mathcal{L}(r_1) \circ \mathcal{L}(r_2)$	$\mathcal{L}(r_1 \cap r_2) = \mathcal{L}(r_1) \cap \mathcal{L}(r_2)$
	$\mathcal{L}(r^+) = \mathcal{L}(r)^+$	
$\mathcal{F}(\lambda) = \emptyset$	$\mathcal{F}(r_1 \cdot r_2) = \mathcal{F}(r_1) \cup (\mathcal{L}(r_1) \cdot \mathcal{F}(r_2))$	$\mathcal{F}(r_1 \cup r_2) = \mathcal{F}(r_1) \cup \mathcal{F}(r_2)$
$\mathcal{F}(b) = \varepsilon$	$\mathcal{F}(r_1 \circ r_2) = \mathcal{F}(r_1) \cup (\mathcal{L}(r_1) \circ \mathcal{F}(r_2))$	$\mathcal{F}(r_1 \cap r_2) = \mathcal{F}(r_1) \cap \mathcal{F}(r_2)$
	$\mathcal{F}(r^+) = \mathcal{L}(r)^* \cdot \mathcal{F}(r)$	
$\mathcal{I}(\lambda) = \emptyset$	$\mathcal{I}(r_1 \cdot r_2) = \mathcal{I}(r_1) \cup (\mathcal{L}(r_1) \cdot \mathcal{I}(r_2))$	$\mathcal{I}(r_1 \cup r_2) = \mathcal{I}(r_1) \cup \mathcal{I}(r_2)$
$\mathcal{I}(b) = \emptyset$	$\mathcal{I}(r_1 \circ r_2) = \mathcal{I}(r_1) \cup (\mathcal{L}(r_1) \circ \mathcal{I}(r_2))$	$\mathcal{I}(r_1 \cap r_2) = \mathcal{I}(r_1) \cap \mathcal{I}(r_2)$
	$\mathcal{I}(r^+) = (\mathcal{L}(r)^* \cdot \mathcal{I}(r)) \cup (\mathcal{L}(r) \setminus \{\varepsilon\})^\omega$	

Fig. 5 The language $\mathcal{L}(r)$, the language of proper prefixes $\mathcal{F}(r)$ and the loop language $\mathcal{I}(r)$ of a SERE r [21, 23, 44]

- $w \models r! \iff \exists j < |w|$ s.t. $w(0..j) \in \mathcal{L}(r)$
- $w \models r \iff$ either $w \models r!$ or $w \in \mathcal{I}(r) \cup \mathcal{F}(r) \cup \{\varepsilon\}$

Fig. 6 The truncated semantics of strong and weak SEREs, as proposed by [21] and adopted by PSL 1850–2010 [44]; see also [23]

ple, the safety component of $[p \text{ U } \text{FALSE}]$ is simply FALSE, while its weak component is $[p \text{ W } \text{FALSE}]$, i.e., G p.

A formula φ is semantically weak if $[\![\varphi_w]\!] = [\![\varphi]\!]$ (where φ_w is defined in Theorem 6). The notion of semantic weakness captures exactly the set of "good" safety properties—those that are computationally easy to verify, as shown by Theorem 19 in Sect. 24.4.3.

Defining the semantics is not enough, of course. We also need to supply an implementation. Automata construction for LTL augmented with strong regular expressions via suffix implication is given in [9, 13, 14]. Implementation for the enhancement with weak regular expressions as well is given in [9, 13]. Special treatments of subsets thereof are given in [10, 11].

24.3 Clocks and Sampling

In this section, we present the clock operator (@). We begin with hardware clocks as a motivating example, and in Sect. 24.3.2 present an example where the @ operator is used as a time-sampling abstraction, separate from the notion of a hardware clock.

24.3.1 Hardware Clocks

Synchronous hardware designs are based on a notion of discrete time, in which a *clock signal* causes memory elements (flip-flops or latches) to transition from one state to the next. The time from one transition until just before the next is termed a

clock cycle. In the early days of hardware model checking, designs typically had a single clock, and tools that built the model from the source code (written in some Hardware Description Language, or HDL) would abstract away the clock cycle, assuming that a clock cycle corresponded to a single step in time of the model. Thus, the following LTL formula:

$$\mathrm{G}(p \rightarrow \mathrm{X}\, q) \tag{16}$$

was used to express the property "globally, if p then *at the next clock cycle*, q".

Modern hardware designs, however, are typically based on multiple clocks. In such a design, for instance, some memory elements may be clocked with *clka*, while others are clocked with *clkb*. In such a case, clock cycles cannot be considered to be atomic—the clock cycles of *clka* and *clkb* might overlap, and the nature of their interaction affects the behavior of the design in important ways. Thus tools that build the model cannot abstract away the clock cycle, and it becomes necessary for the formula to mention the clock signal explicitly. Another complication is that clocking is often done on the *edge* of a clock, resulting in what is termed an *edge-triggered* design. A *positive edge* means a point in time when the clock has just risen from 0 to 1, and a *negative edge* means a point in time when the clock has just fallen from 1 to 0.

The User's Point of View Consider the property "globally, if p at a positive edge of clock *clka*, then at the next positive edge of *clka*, q". In LTL, this is:

$$\begin{aligned} &\mathrm{G}\,(\neg clka \rightarrow \mathrm{X}\big((clka \wedge p) \\ &\quad \rightarrow \mathrm{X}\big[\big(\neg(\neg clka \wedge\ \mathrm{X}\, clka)\big)\ \mathrm{W}\ \big(\neg clka \wedge\ \mathrm{X}\,(clka \wedge q)\big)\big]\big) \end{aligned} \tag{17}$$

Using suffix implication and the goto operator (see Sect. 24.2.2.2), we can make it slightly more readable, as follows:

$$\mathrm{G}\,\big(\big((clka \wedge p)[\rightarrow] \cdot \neg clka[\rightarrow] \cdot clka[\rightarrow]\big) \mapsto q\big) \tag{18}$$

However, taking the clock signal into consideration in each formula quickly becomes unwieldy. Thus the user would like a clock operator, allowing the behavior relative to the clock to be expressed more directly as, for instance:

$$\mathrm{G}(p \rightarrow \mathrm{X}\, q)@(posedge\ clka) \tag{19}$$

Semantic Issues Intuitively, the purpose of the clock operator is to define a projection onto those letters where the clock holds (or transitions, in the case of edge-triggered designs). For example, consider the *trace* shown in Fig. 7. The hardware designer understands the x-axis as time; the value of each signal is indicated by a *waveform*—low indicates a value of 0, and high indicates a value of 1. Formally, we view each point in time as a letter from the alphabet 2^P, mapping the atomic propositions p, q, etc. to either 0 or 1. The projection of the trace onto those points in time where *clka* has a positive edge results in a trace consisting of three points in time, shown shaded in the figure. Formula (19) holds on the full trace because Formula (16) holds on the projection.

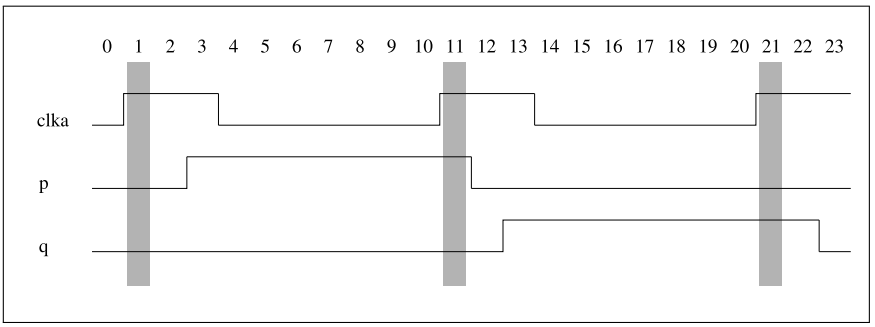


Fig. 7 A trace, showing time on the x-axis. The shaded points in time show the projection of the trace onto the positive edge of *clka*

When we turn to formalize this intuition, we are confronted with two issues. First, hardware clocks do not "accumulate". That is, we want a nested clock operator to have the effect of "changing the projection", rather than further projecting the projected word. This means that while projection is a useful intuition, we cannot simply define that $w \models \varphi@c \iff w' \models \varphi$, where w' is the projection of w onto those letters where c holds, because doing so would mean that we lose the ability to later "change the projection" of the original word w.

Second, we must define what happens if the clock "stops ticking". For example, on an infinite word w such that c holds on $w(0)$ but on no other letter, do we want $(\mathrm{X}\,\varphi)@c$ to hold or not? Thus, the addition of clocks introduces problems similar to those of defining LTL semantics for finite words. And not only may the projection of an infinite word be finite, it may be empty as well.

Early attempts to deal with these issues [6, 18] defined two kinds of clocks, strong and weak, which behaved differently on finite and empty words. Doing so resulted in semantics that suffered from various weaknesses, for instance that a liveness formula may hold on some word w, but not on an extension ww', or that the formula $(\mathrm{F}\ p) \wedge (\mathrm{G}\ q)$ cannot be satisfactorily clocked for a finite word.

The semantics of [26], adopted by PSL and SVA and shown in Fig. 8, solves these problems. In it, the issues of finite and empty words are cleanly separated from the clock operator, whose only role is to define a projection, and nesting of clock operators has the effect of changing the projection. The issue of the semantics on a finite or empty projection is solved by taking the strength from the formula itself, and to this end, there are strong and weak versions of every operator and of Boolean expressions. While the idea of strong and weak clocks has disappeared from PSL and SVA, the issues raised by a clock that stops ticking are manifested in the concept of a truncated path, and dealt with by the truncated semantics discussed in Sect. 24.4.

Note that the semantics are defined for $\varphi@c$, where c is a Boolean expression. Edge-triggered designs are supported by defining that $posedge\ clk$ is equivalent to $ended(\neg clk^* \cdot clk)$ and similarly that $negedge\ clk$ is equivalent to $ended(clk^* \cdot \neg clk)$ (see Sect. 24.2.2.5).

Let f, f_1, and f_2 be formulas in LTL extended with the clock operator @, let b and c be Boolean expressions, and let w be a word over $\Sigma = 2^P$. For finite word w, let $w \stackrel{c}{\models}$ tick denote that $w \in \mathcal{L}(\neg c^* \cdot c)$.

- $w \stackrel{c}{\models} b \iff \forall j < |w|$ s.t. $w(0..j) \stackrel{c}{\models}$ tick , $w(j) \models b$
- $w \stackrel{c}{\models} b! \iff \exists j < |w|$ s.t. $w(0..j) \stackrel{c}{\models}$ tick and $w(j) \models b$
- $w \stackrel{c}{\models} \neg f \iff w \stackrel{c}{\not\models} f$
- $w \stackrel{c}{\models} f_1 \wedge f_2 \iff w \stackrel{c}{\models} f_1$ and $w \stackrel{c}{\models} f_2$
- $w \stackrel{c}{\models} \mathrm{X}! f \iff \exists j < k < |w|$ s.t. $w(0..j) \stackrel{c}{\models}$ tick and $w(j+1..k) \stackrel{c}{\models}$ tick and $w(k..) \stackrel{c}{\models} f$
- $w \stackrel{c}{\models} [f_1 \mathrm{U} f_2] \iff \exists k < |w|$ s.t. $w(k) \models c$ and $w(k..) \stackrel{c}{\models} f_2$ and $\forall j < k$ s.t. $w(j) \models c$, $w(j..) \stackrel{c}{\models} f_1$
- $w \stackrel{c}{\models} f@c_1 \iff w \stackrel{c_1}{\models} f$

Fig. 8 The strengthless clock [26] adopted by PSL and SVA, where the "initial" clock context is defined to be $c = \mathrm{TRUE}$. The semantics of clocked regular expressions are in the same spirit, but not shown for brevity. The interested reader is referred to [43–46]

In the following, let $\models$ denote the traditional semantics of LTL, augmented in the obvious way to support strong and weak next operators and strong and weak Boolean expressions on finite and empty as well as infinite words.

Theorem 7 ([26])[1] *Let f be a formula in LTL augmented with strong and weak next operators and strong and weak propositions, let c be a Boolean expression and w an infinite, finite, or empty word. Let $w|_c$ denote the word obtained from w after leaving only the letters that satisfy c, and let $\stackrel{c}{\models}$ be as defined in Fig. 8. Then*

$$w \stackrel{c}{\models} f \quad \textit{if and only if} \quad w|_c \models f$$

Note that Theorem 7 refers to unclocked formulas, and thus shows that intuition regarding a projection holds for singly clocked formulas. Recall that for multiply clocked formulas, we wanted (and achieved) something different than a projection.

The clock operator does not add expressive power.

Theorem 8 ([26]) *The logic consisting of LTL plus the @ operator is as expressive as LTL.*

The proof of Theorem 8 is by rewriting, thus direct treatment of the @ operator is not required of a tool. However, it can be advantageous to do so, as shown by [55].

In LTL, $[f \mathrm{U} g]$ can be defined as a least solution of the equation $S = g \vee (f \wedge \mathrm{X}! S)$. In the semantics of Fig. 8, there is a fixed point characterization if f and g are themselves unclocked, because $[f \mathrm{U} g]@c \equiv (\mathrm{TRUE}! \wedge g) \vee (f \wedge \mathrm{X}![f \mathrm{U} g])@c$. But if f and g contain clock operators, this equivalence no longer holds [26].

[1] Theorems 7 and 8 were stated and proved in [26]. Later, automated proofs of these theorems were obtained after an embedding of the semantics of PSL into HOL [34].

Let f be a formula in LTL extended with the clock operator @, and let w be a word over $\Sigma = 2^P$. For finite word w, let w *is m clock ticks*, for $m > 0$, denote that $w \in \mathcal{L}((\neg c^* \cdot c)[^*m])$. Then, for $i \geq 0$:

- $w \models^c \mathrm{X}!^i\ f \iff \exists j < |w|$ s.t. $w(0..j)$ is $i+1$ clock ticks of c and $w(j..) \models^c f$

Fig. 9 Generalizing the *next* operator [29], adopted by PSL 1850–2010 [44] and SVA 1800–2009 [46]

As shown in [29], this can be fixed by adding an *alignment operator*, $\mathrm{X}!^0$, that moves to the closest clock tick. The semantics of the alignment operator, adopted by PSL and SVA and shown in Fig. 9, gives the following fixed point characterization of until under a clocked semantics.

Theorem 9 ([29]) *Let f and g be formulas of LTL augmented with the clock operator. Then*:

$[f\ \mathrm{U}\ g]$ *is a least solution of the equation* $S = \mathrm{X}!^0\ (g \vee (f \wedge \mathrm{X}!\, S))$
$[f\ \mathrm{W}\, g]$ *is a greatest solution of the equation* $S = \mathrm{X}^0\ \ (g \vee (f \wedge \mathrm{X}\ \ S))$.

24.3.2 Using a Clock as a Time-Sampling Abstraction

The clock operator is a time-sampling abstraction, and as such has uses other than representing a hardware clock. For example, consider the property that consecutive writes (signal $write$ is asserted) cannot both be high-priority writes (signal $high$ is asserted). Without the clock operator, this can be expressed as follows

$$\mathrm{G}\big((write \wedge high) \rightarrow\ \mathrm{X}\ \big[\neg write\ \mathrm{W}\ (write \wedge \neg high)\big]\big) \tag{20}$$

However, that is quite cryptic. Using the clock operator, we can express it more simply as:

$$\mathrm{G}(high \rightarrow\ \mathrm{X}\ \neg high)@write \tag{21}$$

24.4 Hardware Resets and Other Sources of Truncated Paths

A *path* of a model M with transition relation R is a finite or infinite sequence of states $(s_0, s_1, \ldots, s_n)$ or $(s_0, s_1, \ldots)$, such that each successive pair of states (s_i, s_{i+1}) is an element of R. A path of model M is maximal if either it is infinite, or the last state of the path has no successor in M.

Traditionally, the semantics of temporal logic is defined over infinite words corresponding to infinite paths in the model (see for example Chap. 2), and indeed hardware is a reactive system, whose paths are intrinsically infinite. It turns out, however, that hardware designers do find themselves needing to reason over finite

paths, specifically, over *truncated paths*—paths that are finite, but not necessarily maximal.

The need to reason over truncated paths arises in several different contexts. Incomplete methods of verification, such as bounded model checking or dynamic verification, naturally reason over truncated paths. A hardware reset can also be understood as truncating a path, because anything that comes after should not affect the truth value of the property on the infinite path. Finally, hardware clocks are related to truncated paths, because a clock that stops ticking gives that the projection of an infinite path may be finite. Early definitions of a clock operator struggled with this issue, which was solved by the truncated semantics, leaving the clock operator to define a projection without worrying about whether or not the clock ticks infinitely often.

Here, we start with a presentation of hardware resets in Sect. 24.4.1. In Sect. 24.4.2 we move to a discussion of truncated paths arising from other sources, and then show that hardware resets are a particular case of truncated paths. Section 24.4.3 discusses the surprising relation of truncated paths to the classification of safety formulas.

24.4.1 Hardware Resets

The User's Point of View Consider the property that if a high-priority request is received (signal hi_rq is asserted) then one of the next two grants (assertion of signal gnt) will be to the high-priority destination (signal hi_dt is asserted). In LTL, this can be expressed as:

$$\mathrm{G}\big(hi\_rq \rightarrow \big[\neg gnt \ \mathrm{W}\ \big(gnt \wedge \big((hi\_dt) \vee \ \mathrm{X}\big[\neg gnt \ \mathrm{W}\ (gnt \wedge hi\_dt)\big]\big)\big)\big]\big) \quad (22)$$

although the user will usually prefer to use the suffix implication operator to obtain the much simpler form:

$$\mathrm{G}\big(hi\_rq \mapsto \big(gnt[\rightarrow 1..2] \circ hi\_dt\big)\big) \quad (23)$$

Now, if the reset signal (rst) is to be taken into consideration, such that when signal rst is asserted, all outstanding requirements are cancelled, and reckoning begins again at the next deassertion of rst, then Formula (22) must be modified as follows:

$$\begin{aligned}&\mathrm{G}\big((hi\_rq \wedge \neg rst) \rightarrow \\ &\quad \big[\neg gnt \ \mathrm{W}\ \big(rst \vee \big(gnt \wedge \big(hi\_dt \vee \ \mathrm{X}\big[\neg gnt \ \mathrm{W}\ \big(rst \vee (gnt \wedge hi\_dt)\big)\big]\big)\big)\big)\big]\big)\end{aligned} \quad (24)$$

This is quite cumbersome, and while modifying Formula (23) in a similar manner is less verbose:

$$\mathrm{G}\big(\big(hi\_rq \circ gnt[\rightarrow 1..2]\big) \cap \big((\neg rst)^*\big) \mapsto hi\_dt \quad (25)$$

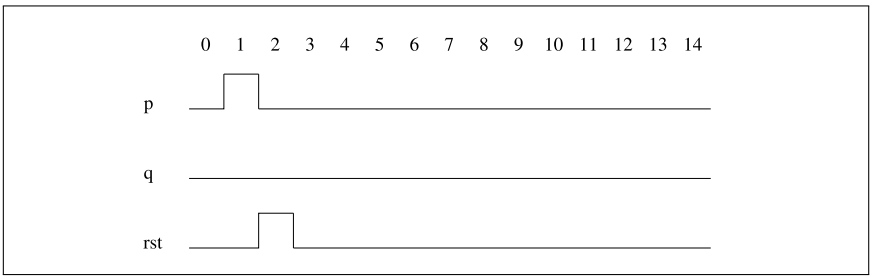


Fig. 10 Weak until vs. reset. Formula (28) does not hold, but Formula (29) does

It requires moving part of the formula from the right- to the left-hand side of the suffix implication, which could be problematical if the original LTL version had used strong until. The user would like an easier way, so that Formulas (22) and (23) can be reset simply like this:

$$\mathrm{G}\big(hi\_rq \rightarrow \big[\neg gnt \ \mathrm{W}\ \big(gnt \wedge \big(hi\_dt \vee \ \mathrm{X}\big[\neg gnt \ \mathrm{W}\ (gnt \wedge hi\_dt)\big]\big)\big)\big]\big) \ \text{RESET}\ rst \tag{26}$$

$$\mathrm{G}\big(hi\_rq \mapsto \big(gnt[\rightarrow 1..2] \circ hi\_dt\big)\big) \ \text{RESET}\ rst \tag{27}$$

Semantic Issues Intuitively, the desired semantics of the reset operator is that up until the reset condition, "nothing has yet gone wrong." So, informally, we want $w \models \varphi \ \text{RESET}\ b$ to hold iff φ holds on w truncated at the first occurrence of b. Before continuing, it is instructive to compare the desired semantics of the reset operator with the semantics of the weak until operator. Consider

$$\big[(p \rightarrow \mathrm{XXX}\, q) \ \mathrm{W}\ rst\big] \tag{28}$$

and

$$\big(\mathrm{G}(p \rightarrow \mathrm{XXX}\, q)\big) \ \text{RESET}\ rst \tag{29}$$

on a word w such that p holds on (and only on) $w(1)$, q holds on no letter, and rst holds on (and only on) $w(2)$. This is illustrated in Fig. 10. Then Formula (28) does not hold on w, but we want that Formula (29) does hold on w. Thus the rst in $(\mathrm{G}\ \varphi)\ \text{RESET}\ rst$ can be thought of as "canceling future obligations of φ," as opposed to $[\varphi\ \mathrm{W}\ rst]$, in which the "obligations" of φ may extend beyond the first occurrence of rst.

It is tempting to define that $\varphi \ \text{RESET}\ b$ holds on w if φ holds on w or if there exist words u and v and letter ℓ such that b holds on ℓ, $w = u\ell v$, and u has an extension on which φ holds. However, such a definition does not give us what we want for reasons similar to those discussed in Sect. 24.2.3 regarding the definition of weak regular expressions. For example, we would like $[busy\ \mathrm{U}\ \text{FALSE}]\ \text{RESET}\ b$ to hold on a word where b holds on some letter and $busy$ holds on every letter up until the letter where b holds, but a definition that looks for an extension on which $[busy\ \mathrm{U}\ \text{FALSE}]$ holds does not give us that. Furthermore, the complexity of model checking using such a definition is non-elementary [4].

Let φ and ψ be formulas of LTL extended with the RESET operator, let a and r be Boolean expressions, and let w be a word over $\Sigma = 2^P$.

- $\langle w, a, r\rangle \models p \iff w(0) \Vdash a \vee (p \wedge \neg r)$
- $\langle w, a, r\rangle \models \neg\varphi \iff \langle w, r, a\rangle \not\models \varphi$
- $\langle w, a, r\rangle \models \varphi \wedge \psi \iff \langle w, a, r\rangle \models \varphi$ and $\langle w, a, r\rangle \models \psi$
- $\langle w, a, r\rangle \models \mathrm{X}\,\varphi \iff w(0) \Vdash a$ or both $w(0) \not\Vdash r$ and $\langle w(1..), a, r\rangle \models \varphi$
- $\langle w, a, r\rangle \models [\varphi\ \mathrm{U}\ \psi] \iff \exists k < |w|$ s.t. $\langle w(k..), a, r\rangle \models \psi$, and $\forall j < k,\ \langle w(j..), a, r\rangle \models \varphi$
- $\langle w, a, r\rangle \models \varphi\ \mathrm{RESET}\ b \iff \langle w, a \vee (b \wedge \neg r), r\rangle \models \varphi$

Fig. 11 The reset semantics [4, 6]

The solution of [6] was to add two contexts, the *accept* condition and the *reject* condition. The resulting semantics is shown in Fig. 11, and following [4], we term these the *reset* semantics. A formula φ holds in a model if $\langle w, \mathrm{FALSE}, \mathrm{FALSE}\rangle \models \varphi$ for every word in the model. The complexity of the reset semantics is no different than that of LTL, as stated by Theorem 10.

Theorem 10 ([4]) *The satisfiability and model-checking problems for LTL plus the reset operator under the reset semantics are PSPACE-complete.*

The reset operator does not add expressive power. Automata construction for the reset operator, as well as rewrite rules for translating LTL plus the reset operator into LTL are given by [4].

Theorem 11 ([4]) *The logic consisting of LTL plus the reset operator is as expressive as LTL.*

More insight into the reset semantics can be found in the next subsection, which shows two equivalent statements of the reset semantics.

24.4.2 Other Sources of Truncated Paths

The User's Point of View Model checking is a tool, but not the only tool. Users want a specification language that is equally relevant to every verification method in their toolbox. Consider the property "every request must receive a grant, and once asserted, the request signal must stay asserted until it receives its grant". Typically we would specify this in LTL as follows:

$$\mathrm{G}\big(request \rightarrow \mathrm{X}\,[request\ \mathrm{U}\ grant]\big) \tag{30}$$

However, when using an incomplete method of verification, such as bounded model checking or simulation, the user has a decision to make. Is it possible that the verification run will end in between a request and its grant, so that

$$\mathrm{G}\big(request \rightarrow \mathrm{X}\,[request\ \mathrm{W}\ grant]\big) \tag{31}$$

should be used instead?

The answer does not depend on the design under verification, but it does depend on properties of the verification method. For instance, some simulation tests are designed to continue until correct output can be confirmed, and then the answer is that Formula (30) should be used. On the other hand, some tests have no "opinion" on the correct length of a test, and they may end at any time. In such a case, Formula (30) might result in a false negative, and Formula (31) should be used instead.

The answer might depend on properties of the verification method, but designers want the specification to specify the design, and be reusable no matter the method of verification. Thus it is desirable for a specification language to provide a means to distinguish between properties of the design and properties of the verification.

Semantic Issues Distinguishing between properties of the design and properties of the verification can be achieved by defining separate *views* of the same formula, and distinguishing between maximal and non-maximal, or *truncated*, words [24]. The formula itself describes the design, while each view is useful in a different verification method. The views differ only on finite words, and only in cases in which there is doubt whether the formula holds on the original, possibly unknown, untruncated word. For instance, consider the formula F p on a truncated word such that p does not hold on any letter, or the formula G q on a truncated word such that q holds on every letter. In both cases it is impossible to know whether or not the formula holds on the original, untruncated, word.

In such cases, in which there is doubt as to whether the formula holds on the original word, the truncated semantics, shown in Fig. 12, define that it holds in the *weak view*, does not hold in the *strong view*, and holds in the *neutral view* iff it holds in the traditional LTL semantics on finite words (this is equivalent to considering the word to be maximal rather than truncated). Note that Fig. 12 does not show three separate semantics, but rather a single semantics, in which the view is a context. Thus $w \models^{-} \varphi$, $w \models \varphi$, and $w \models^{+} \varphi$ should be understood as shorthand for $\langle w, \text{weak} \rangle \models \varphi$, $\langle w, \text{neutral} \rangle \models \varphi$, and $\langle w, \text{strong} \rangle \models \varphi$, respectively.

Let us now return to the example represented by Formulas (30) and (31). The specification should describe the design, thus Formula (30) is the formula that should be used. In the case of a test that was designed to continue until correct output can be confirmed, Formula (30) can be checked under the neutral view. In the case that the test may end at any time, Formula (30) can be checked under the weak view. Note that most formulas do not hold under the strong view on finite words, because for any φ, Gφ does not hold under the strong view on such a word. Thus the main purpose of the strong view is to serve as a dual to the weak.

As shown in [24], the truncated semantics has the property that the strong view is stronger than the neutral view, which is in turn stronger than the weak view, as stated by Theorem 12 below. It also supports the intuition that φ holds weakly on w if up till now nothing "has gone wrong," and thus holds as well on any prefix of w, and that φ holds strongly on w if "all future obligations have been met," and thus holds as well on any extension of w. This is stated by Theorem 13 below.

Let φ and ψ be LTL formulas, let b be a Boolean expression, and let w be a word over $\Sigma = 2^P$. Allow "overflow" of indices: for $k \geq |w|$, let $w(k..) = \varepsilon$.

holds weakly: For w over 2^P such that $|w| \geq 0$,

- $w \models^{-} b \iff |w| = 0$ or $w(0) \Vdash b$
- $w \models^{-} \neg\varphi \iff w \not\models^{+} \varphi$
- $w \models^{-} \varphi \wedge \psi \iff w \models^{-} \varphi$ and $w \models^{-} \psi$
- $w \models^{-} \mathrm{X}!\,\varphi \iff w(1..) \models^{-} \varphi$
- $w \models^{-} [\varphi\ \mathrm{U}\ \psi] \iff \exists k$ such that $w(k..) \models^{-} \psi$ and for every $j < k$, $w(j..) \models^{-} \varphi$

holds neutrally: For w over 2^P such that $|w| > 0$,

- $w \models b \iff w(0) \Vdash b$
- $w \models \neg\varphi \iff w \not\models \varphi$
- $w \models \varphi \wedge \psi \iff w \models \varphi$ and $w \models \psi$
- $w \models \mathrm{X}!\,\varphi \iff |w| > 1$ and $w(1..) \models \varphi$
- $w \models [\varphi\ \mathrm{U}\ \psi] \iff \exists k < |w|$ such that $w(k..) \models \psi$ and for every $j < k$, $w(j..) \models \varphi$

holds strongly: For w over 2^P such that $|w| \geq 0$,

- $w \models^{+} b \iff |w| > 0$ and $w(0) \Vdash b$
- $w \models^{+} \neg\varphi \iff w \not\models^{-} \varphi$
- $w \models^{+} \varphi \wedge \psi \iff w \models^{+} \varphi$ and $w \models^{+} \psi$
- $w \models^{+} \mathrm{X}!\,\varphi \iff w(1..) \models^{+} \varphi$
- $w \models^{+} [\varphi\ \mathrm{U}\ \psi] \iff \exists k$ such that $w(k..) \models^{+} \psi$ and for every $j < k$, $w(j..) \models^{+} \varphi$

Fig. 12 The truncated semantics of LTL [24]

Theorem 12 (Strength Relation Theorem [24]) *Let w be a non-empty word. Then*:

- $w \models^{+} \varphi \Longrightarrow w \models \varphi$
- $w \models \varphi \Longrightarrow w \models^{-} \varphi$

Theorem 13 (Prefix/Extension Theorem [24])

- $v \models^{+} \varphi \iff \forall w \succeq v : w \models^{+} \varphi$
- $v \models^{-} \varphi \iff \forall u \preceq v : u \models^{-} \varphi$

The reset operator, discussed in detail in the previous subsection, is easily stated in the truncated semantics, as a reset can be understood as truncating an infinite word, and moving to the weak view. Figure 13 shows the formal statement of the truncated semantics of the reset operator.

The reset semantics and the truncated semantics accomplish their goal by restating the semantics of every operator. This complicates things considerably in a temporal logic that has many core operators (as is the case in PSL and SVA). Using ideas from [22] discussed in Sect. 24.2.3 one can restate the semantics more succinctly without requiring a restatement of the semantics of each operator. This approach makes use of the special letters $\top$ and $\bot$. Recall that $\top$ satisfies all Boolean expres-

Let φ be a formula of LTL extended with the RESET operator, let b be a Boolean expression, and let w be a word over $\Sigma = 2^P$.

- $w \models^{-} \varphi \text{ RESET } b \iff$ either $w \models^{-} \varphi$ or $\exists j < |w|$ s.t. $w(j) \models b$ and $w(0..j-1) \models^{-} \varphi$
- $w \models \varphi \text{ RESET } b \iff$ either $w \models \varphi$ or $\exists j < |w|$ s.t. $w(j) \models b$ and $w(0..j-1) \models^{-} \varphi$
- $w \models^{+} \varphi \text{ RESET } b \iff$ either $w \models^{+} \varphi$ or $\exists j < |w|$ s.t. $w(j) \models b$ and $w(0..j-1) \models^{-} \varphi$
- All other LTL operators are as usual.

Fig. 13 The truncated semantics of LTL extended with the RESET operator [24]

Let φ be a formula of LTL extended with the RESET operator, let b be a Boolean expression. Let $\top$ and $\bot$ be special letters such that $\top \models b$ and $\bot \not\models b$ for any b (including TRUE and FALSE). Let w be a word over $\Sigma = 2^P \cup \{\top, \bot\}$. Let $\overline{w}$ denote the word obtained from w by switching every $\top$ with $\bot$ and vice versa.

- $w \models \neg\varphi \iff \overline{w} \not\models \varphi$
- $w \models \varphi \text{ RESET } b \iff$ either $w \models \varphi$ or $\exists j < |w|$ such that $w(j) \models b$ and $w(0..j-1)\top^{\omega} \models \varphi$
- All other LTL operators are as usual.

Fig. 14 $\top,\bot$ approach to the truncated semantics [25], adopted by PSL 1850–2005 [43] and SVA [45, 46]

sions including FALSE while $\bot$ satisfies no Boolean expression, not even TRUE. The resulting semantics, referred to as the $\top,\bot$ approach, is given in Fig. 14.

Theorem 14 ([24, 25]) *Let φ be a formula of LTL extended with the reset operator. Let w be a word over $\Sigma = 2^P$. Then $\langle w, \text{FALSE}, \text{FALSE}\rangle \models \varphi$ in the reset semantics iff $w \models \varphi$ in the truncated semantics iff $w \models \varphi$ in the $\top,\bot$ approach to the truncated semantics.*

As mentioned in Sect. 24.2.3, the $\top$, $\bot$ approach breaks when regular expression intersection and fusion are included [21, 22]. Thus the semantics of PSL 1850–2010 [44] uses a version of the truncated semantics that supports SEREs, defined in [21], characterized in [23], and shown in Figs. 5 and 6.

24.4.3 The Truncated Semantics and Classification of Safety Formulas

A *bad prefix* of a safety formula φ is a finite word w all of whose infinite extensions violate φ. An *informative prefix*, defined syntactically in [50] in order to classify safety properties, is, loosely speaking, a bad prefix that provides enough information to "explain" why φ does not hold. For instance, an informative prefix of $\varphi = \text{X!X!}p$ would be a word of length at least 3 where p does not hold on the third letter.

The formula $\psi = \mathrm{FG}\, p \wedge \mathrm{FG} \neg p$ has no informative prefixes, since no finite prefix is long enough to "explain" why it fails. This, despite the fact that ψ is a contradiction, thus every finite word is a bad prefix of it.

The motivation for the classification of safety formulas is complexity. For general LTL properties, the automata-theoretic approach to model checking [53, 66, 72] builds the product of the design under verification and a Büchi automaton for the negation of the property and checks for emptiness. For safety properties it suffices to work with automata on finite words (see Chap. 2), and an automaton on finite words recognizing bad prefixes can also be used by simulation, simply by translating it into a checker coded in some Hardware Description Language [1]. Model checking with a finite automaton rather than a Büchi automaton should be easier since it replaces a search for fair cycles with an invariant check. However, the complexity results tell a different story. For an arbitrary LTL formula φ, the equivalent Büchi automaton is of size exponential in $|\varphi|$, while for a safety LTL formula φ, the size of an automaton recognizing its bad prefixes is doubly exponential in $|\varphi|$.

Theorem 15 ([72]) *Let φ be an LTL formula of size n. Then there exists a nondeterministic Büchi automaton of size $2^{O(n)}$ recognizing $[\![\varphi]\!]$.*

Theorem 16 ([50]) *Let φ be an LTL formula of size n. The size of an NFA recognizing the bad prefixes of φ is $2^{2^{O(n)}}$ and $2^{2^{\Omega(\sqrt{n})}}$.*

As shown by [50], if we are willing to limit ourselves to recognizing all informative, rather than bad, prefixes, we return to a single exponent.

Theorem 17 ([50]) *Let φ be a safety LTL formula of size n. There exists an NFA of size $2^{O(n)}$ recognizing all the informative prefixes of φ.*

The question then becomes whether we can use the automaton for informative prefixes instead of the one for bad prefixes. As noted in [50], it suffices to recognize a single bad prefix for each violating word. Thus they classify safety properties according to whether all, some, or none of its bad prefixes are informative. A property is *intentionally safe* if all of its bad prefixes are informative. A property φ is *accidentally safe* if every computation that violates it has an informative prefix. A property is *pathologically safe* if there is a computation that violates φ and has no informative prefix. It follows from Theorem 17 that for non-pathological formulas, an automaton of exponential (rather than doubly exponential) size exists that recognizes at least one bad prefix for each violating word, matching our intuition that (non-pathological) safety formulas are easier to check than general LTL formulas.

The motivation for the classification of safety formulas was very different from the motivation for the truncated semantics, but it turns out that the two are related in an interesting way. While [50] defines an informative prefix syntactically, the truncated semantics provides a semantic definition:

Theorem 18 ([24]) *Let φ be an LTL formula and w a finite non-empty word. Then w is an informative prefix for φ iff $w \not\models^{-} \varphi$.*

Let b be a Boolean expression and r a regular expression. Then RLTL$^{\mathrm{LV}}$ consists of the formulas defined by the following grammar:

$$\varphi ::= b \mid \varphi \wedge \varphi \mid \mathrm{X}\,\varphi \mid (b \wedge \varphi) \vee (\neg b \wedge \varphi) \mid \left[(b \wedge \varphi)\ \mathrm{W}\ (\neg b \wedge \varphi)\right] \mid r \mapsto \varphi$$

Fig. 15 RLTL$^{\mathrm{LV}}$, a syntactic subset of LTL extended with suffix implication (see Sect. 24.2.1)

Furthermore, the notion of semantic weakness discussed in Sect. 24.2.3 and captured by the truncated semantics provides a semantic characterization of the set of safety properties that are computationally easy to verify:

Theorem 19 ([22, 23]) *Let φ be an LTL formula. Then φ is semantically weak iff it is non-pathologically safe.*

24.5 The Simple Subset

In Sect. 24.4.3 we saw a subset of safety formulas that are computationally easy to verify, relative to all of LTL. Finding a syntactic subset for which simulation and model checking are guaranteed to be even more efficient was the intention of the *simple subset* of PSL, defined in [43, 44]. Figure 15 shows RLTL$^{\mathrm{LV}}$, a syntactic subset of LTL extended with suffix implication (see Sect. 24.2.1), defined in [10] and subsuming the fragment of the simple subset that uses only weak operators. The *lv* of RLTL$^{\mathrm{LV}}$ stands for *linear violation*, and as we shall see below, formulas of RLTL$^{\mathrm{LV}}$ can be model checked using an NFA that is linear in the size of the formula, rather than exponential as for arbitrary non-pathologically safe formulas.

Intuitively, time flows from left to right through formulas in the simple subset, and syntactically every temporal binary operator has only one non-temporal operand. In this respect, the simple subset shares much in common with the subset of RCTL formulas that can be model checked using invariance checking [8] (RCTL is an extension of CTL with suffix implication). It is also reminiscent of the syntactic subset LTL$^{\mathrm{DET}}$ of LTL described by [56], in which every formula that can be expressed in both LTL and ACTL has an equivalent. In fact, the restriction of PSL's simple subset to LTL formulas is a subset of LTL$^{\mathrm{DET}}$.

While the motivation in [56] was expressiveness, it was also shown there that formulas in LTL$^{\mathrm{DET}}$ have a 1-weak Büchi automaton of linear size, where a 1-weak Büchi automaton is a Büchi automaton in which every strongly connected component is a singleton. The structure of 1-weak Büchi automata make them more efficiently checkable than general Büchi automata [12, 60], making formulas in the simple subset more efficiently checkable than formulas not in the subset.

Theorem 20 ([56]) *Let φ be an LTL^{DET} formula of size n. There exists a 1-weak Büchi automaton of size $O(n)$ recognizing $[\![\varphi]\!]$.*

The analogous result for RLTL^{LV} provides the promised efficiency for both simulation and model checking:

Theorem 21 ([10]) *Let φ be an $RLTL^{LV}$ formula of size n. There exists an NFA of size $O(n)$ recognizing the informative prefixes of φ.*

24.6 Quantified and Local Variables

Both PSL and SVA provide quantified and local variables. Quantified variables, discussed in Sect. 24.6.1, are variables in the familiar sense in logic. Local variables, discussed in Sect. 24.6.2, are different—the intuition behind them borrows from programming languages. Intuitively, local variables are a mechanism for both declaring quantified variables and constraining their behavior.

24.6.1 Quantified Variables

The User's Point of View Suppose that we want to check that every read request returns correct data, in a design that uses tagged requests. In such a design, every request gets an identifying number (the *tag*), which is used to identify the corresponding data. Thus we want to check that if there is a read request to address a, tagged with tag t, and the value of the data at address a is d, then the next time that data for tag t appears on the data bus, the value of the data is d. Conceptually, the formula we want to check is that for every value of a, t and d,

$$\begin{aligned}&\mathrm{G}\big(\big((addr = a) \wedge (tag = t) \wedge \big(mem(a) = d\big)\big) \rightarrow \\ &\quad \big[\neg\big(data\_valid \wedge (tag = t)\big)\ \mathrm{W}\ \big(data\_valid \wedge (tag = t) \wedge (data = d)\big)\big]\big)\end{aligned} \tag{32}$$

If an address is 32 bits wide, a tag is 8 bits wide, and data is 128 bits wide, then without quantifiers, we will need to write 2^{168} LTL formulas. Thus the user wants quantified variables, allowing the 2^{168} formulas to be expressed more succinctly in a single formula.

Semantic Issues Formula (32) requires quantification over *rigid* variables—variables whose valuation stays constant over time. Clearly, adding quantification over such variables does not increase the expressive power (though as discussed above it adds much succinctness). Both PSL and SVA support quantification over

rigid variables, allowing the 2^{168} formulas referred to above to be expressed as a single formula, as follows:

$$\begin{aligned}&\forall a \in \mathbb{B}^{32}\ \forall t \in \mathbb{B}^{8}\ \forall d \in \mathbb{B}^{128} \\ &\mathrm{G}\Big(\big((addr = a) \wedge (tag = t) \wedge \big(mem(a) = d\big)\big) \\ &\quad \rightarrow \big[\neg\big(data\_valid \wedge (tag = t)\big)\ \mathrm{W}\ \big(data\_valid \wedge (tag = t) \wedge (data = d)\big)\big]\Big)\end{aligned} \tag{33}$$

Adding quantification over *flexible* variables—variables that may have different values at different time points—increases the expressive power of LTL to that of ω-regular languages. While LTL cannot count (a fact known through [31, 47, 58]), LTL extended with universal/existential quantification over flexible variables, henceforth referred to as QLTL, can. Indeed, the following QLTL formula:

$$\forall t.\big(t \wedge \mathrm{G}(t \leftrightarrow \mathrm{X}\neg t)\big) \rightarrow \mathrm{G}(t \rightarrow p) \tag{34}$$

expresses that p holds at every even position, a property expressible in PSL and SVA using regular expressions, but not expressible in LTL [74] (see also Chap. 2). Note that here t is a variable standing for an atomic proposition whose behavior over time is unknown, whereas rigid variables play the role of constants rather than atomic propositions.

QLTL was introduced in [65]. It was shown in [67] that its expressive power is ω-regular and the complexity of the satisfiability problem is non-elementary. In [48] it was argued that QLTL is also important for reasoning about abstraction refinement methods, and a complete axiomatic proof system for this logic was provided.[2] The QLTL property expressed by Formula (34) can practically be checked by a model checker for plain LTL by introducing an unconstrained auxiliary variable t and checking the formula $(t \wedge \mathrm{G}(t \leftrightarrow \mathrm{X}\,\neg t)) \rightarrow (\mathrm{G}(t \rightarrow p))$. Obviously this is not a general solution, for example if quantifications are nested with alternating quantifiers.

24.6.2 Local Variables

The User's Point of View Consider now the property that every request must receive a unique grant, where n is the maximum number of requests that may be outstanding (i.e., that have not yet received a grant). Designate a request by r and a grant by g, and assume for simplicity that requests and grants are mutually exclusive. Then for $n = 1$, the property can easily be expressed as

$$\mathrm{G}\ \big(r \rightarrow \mathrm{X}[\neg r\ \mathrm{U}\ g]\big) \tag{35}$$

[2] While [48] considers a logic with past operators, [30] enhances the result to a logic without them.

For $n > 1$, we can turn to the power of regular expressions. If we can express the set of shortest counterexamples by a regular expression, then we can assert that that regular expression never occurs. As previously, let r stand for request and g for grant, and let e stand for "else", that is, $\neg r \wedge \neg g$. Then the set of shortest counterexamples for $n = 2$ is the language of the following regular expression:

$$e^* \cdot r \cdot \big(e \cup \big(g \cdot e^* \cdot r\big) \cup \big(r \cdot e^* \cdot g\big)\big)^* \cdot r \cdot e^* \cdot r \tag{36}$$

and for $n = 3$ it is the language of this:

$$e^* \cdot r \cdot \Big(\big(g \cdot e^* \cdot r\big) \cup e \cup \big(r \cdot \big(e \cup \big(r \cdot e^* \cdot g\big)\big)^* \cdot g\big)\Big)^* \cdot$$
$$r \cdot \big(e \cup \big(r \cdot e^* \cdot g\big)\big)^* \cdot r \cdot e^* \cdot r \tag{37}$$

As n grows, the set of shortest counterexamples (and thus our property) turns out to be surprisingly difficult to express. In contrast, it is very easy to construct a non-deterministic finite automaton for a particular n. The user would like the same ease of expression inside the specification language. For instance, she would like to be able to say the following:

$$\text{NEW}(v{=}0)\ \Big(\big((r \vee g)[\rightarrow],\ v{=}(r\ ?\ v{+}{+}\ :\ v{-}{-})\big)^*\ \mapsto\ \big((v \geq 0) \wedge (v \leq n)\big)\Big) \tag{38}$$

where $b[\rightarrow]$ abbreviates $\neg b^* \cdot b$ and $=$ stands for assignment. Formula (38) declares a *local variable* v, controlled from within the regular expression. Its initial value is 0, and it is incremented every time there is a request (r) and decremented every time there is a grant (g). The formula holds iff v stays in the range $0 \leq v \leq n$.

Semantic Issues Intuitively, local variables can be seen as a mechanism for both declaring quantified variables and constraining their behavior. For instance, consider the following formula using local variables:

$$\text{NEW}(i, j)\ \Big((a, i{=}0, j{=}0) \cdot (b, i{+}{+})^* \cdot (c, j{+}{+})^* \cdot (i{=}{=}j)\Big) \mapsto d \tag{39}$$

It states that a sequence consisting of an a followed by some number of b's followed by the same number of c's should be followed by a d. It uses the local variables i and j to make sure the same number of b's and c's are observed. If i and j are unbounded, local variables may increase the expressive power to that of context-free languages. In practice, however only bounded local variables are considered. Restricting attention to bounded local variables, there is no increase in expressiveness over ω-regular languages, but there is an increase in succinctness. Use of local variables was advocated in [61, 64].

When we try to formulate the semantics of a logic with local variables we confront the issue that for a given word w and a given regular expression r we may want more than one value of a local variable in each letter of w. For instance, consider Formula (39) and a word where a holds on the first letter and only on the first letter, and both b and c hold on the second to fifth letters and only on the second to fifth letters. What is the value of i and j on the fifth letter? It could be $i = 2, j = 2$ by

Let V be a set of local variables and let $\mathrm{Z} \subseteq V$. Let D be the domain of the local variables and let $\gamma_1, \gamma_2 \in D^V$ be assignments to the local variables. Let $\gamma_1 \overset{\mathrm{Z}}{\sim} \gamma_2$ (read "γ_1 agrees with γ_2 relative to Z") denote that for every $z \in \mathrm{Z}$ we have that the value of variable z in valuation γ_1 is the same as its value in γ_2. For an expression e, and an enhanced letter $\mathfrak{a}$, the notation $e[\mathfrak{a}|_{\sigma\gamma}]$ provides the value of e with respect to the σ and γ components of $\mathfrak{a}$. The notation $\mathfrak{v} \models_{\mathrm{Z}} r$ means that the good enhanced word $\mathfrak{v}$ tightly models r with respect to controlled variables Z.

- $\mathfrak{v} \models_{\mathrm{Z}} b, x{=}e \iff |\mathfrak{v}| = 1$ and $b[\mathfrak{v}(0)|_{\sigma\gamma}] = \mathrm{T}$ and $\mathfrak{v}(0)|_{\gamma'} \overset{\mathrm{Z}}{\sim} \hat{\gamma}$ where $\hat{\gamma}$ results from $\mathfrak{v}(0)|_{\gamma}$ by assigning $e[\mathfrak{v}(0)|_{\sigma\gamma}]$ to x
- $\mathfrak{v} \models_{\mathrm{Z}} r_1 \cdot r_2 \iff \exists \mathfrak{v}_1, \mathfrak{v}_2$ such that $\mathfrak{v} = \mathfrak{v}_1\mathfrak{v}_2$ and $\mathfrak{v}_1 \models_{\mathrm{Z}} r_1$ and $\mathfrak{v}_2 \models_{\mathrm{Z}} r_2$
- $\mathfrak{v} \models_{\mathrm{Z}} r_1 \cup r_2 \iff \mathfrak{v} \models_{\mathrm{Z}} r_1$ or $\mathfrak{v} \models_{\mathrm{Z}} r_2$
- $\mathfrak{v} \models_{\mathrm{Z}} r_1 \cap r_2 \iff \mathfrak{v} \models_{\mathrm{Z}} r_1$ and $\mathfrak{v} \models_{\mathrm{Z}} r_2$
- $\mathfrak{v} \models_{\mathrm{Z}} \{\mathrm{NEW}(\mathrm{Y})\ r\} \iff \mathfrak{v} \models_{\mathrm{Z}\cup\mathrm{Y}} r$
- $\mathfrak{v} \models_{\mathrm{Z}} \{\mathrm{FREE}(\mathrm{Y})\ r\} \iff \mathfrak{v} \models_{\mathrm{Z}\setminus\mathrm{Y}} r$

Fig. 16 Parts of the PSL semantics for SEREs with local variables as per [20, 44]

matching the b's on the second and third letters and the c's on the fourth and fifth letters; it could also be $i = 3$, $j = 1$ by matching b's on the second to fourth letters and c on the fifth letter. Several other options are possible as well. For the formula to hold d should hold on both the third and fifth letters—the locations where i and j may be equal.

Another issue one confronts when formulating the semantics has to do with the intersection operator. Suppose we want a SERE whose language includes only words where the number of a's between s and e equals the number of b's between s and e. Consider the following formula:

$$\mathrm{NEW}(i)\ \Big((s, i{=}0)\ \cdot\ \Big(\big(\neg a^* \cdot (a, i{+}{+})\big)^*\ \cap\ \big(\neg b^* \cdot (b, i{+}{+})\big)^*\Big)\ \cdot e\Big) \tag{40}$$

Using simple intersection will require the a's and the b's to hold on the same cycles on the words being intersected, which is not what we want.

For this reason, the idea in the first formal definition for a logic augmented with local variables, given in [15, 45], was to control local variables only at the beginning and end of the word. Regular expression semantics is defined with respect to a word w and two contexts L_0 and L_1 standing for the values of local variables at the beginning and end of the word, respectively. It was shown in [40] that this semantics breaks distributivity of union over intersection.

This drawback was addressed in the definition proposed in [20] and adopted by PSL. Parts of the semantics of SEREs with local variables as per [20] are given in Fig. 16. It is argued in [20] that any semantics that try to automatically divide the responsibility for the set of controlled variables between the SERE operands of intersection and union will break distributivity or another basic algebraic property. Instead, the approach was to define the semantics with respect to (1) *good enhanced words* and (2) a set of *controlled local variables*.

The semantics without local variables is defined with respect to words over $\Sigma = 2^P$ where P is the set of atomic propositions. The enhanced alphabet is

$\Sigma \times \Gamma \times \Gamma$ where $\Gamma = D^V$, V is the set of local variables, and D is their domain. Let $\langle \sigma, \gamma, \gamma' \rangle$ be an enhanced letter. The component σ provides a valuation to the atomic propositions; the component γ provides a valuation to the local variables before assignments are executed; and the component γ' provides a valuation to the local variables after assignments are executed. An enhanced word is *good* if the component γ of a given letter is equivalent to the component γ' of the previous letter (if such a letter exists). Thus, in comparison to [15], local variables are watched at every letter of the word rather than just at the beginning and end of a word.

To deal with the problem imposed by the intersection operator, the semantics is defined with respect to a *set of controlled variables* Z which is a subset of the local variables V corresponding to the variables whose valuation should be observed. A controlled variable should change its value according to an assignment given to it, if such was made, and retain its value otherwise. The values of uncontrolled variables do not play a role in the semantics. The control over which variables should be controlled is given to the user by means of two operators $\text{NEW}(x)$ and $\text{FREE}(x)$ which add and remove, respectively, a variable from the set of controlled variables.

Theorem 22 ([20]) *The following properties hold for SEREs extended with local variables using the semantics of* [20]:

- *The operators* $\cup$ *and* $\cap$ *are commutative, and* $\cup$, $\cap$, *and* $\cdot$ *are associative.*
- *The operator* $\cup$ *distributes over* $\cap$ *and* $\cap$ *distributes over* $\cup$.
- $r \equiv (r \cup r) \equiv (r \cap r) \equiv (\lambda \cdot r) \equiv (r \cdot \lambda) \equiv (\text{FALSE} \cup r) \equiv ((\text{FREE}(V)\ \text{TRUE})^* \cap r)$.

The addition of local variables increases the complexity of the model-checking problem from PSPACE to EXPSPACE [15]. This holds for both approaches to the semantics. Automata construction for the semantics of the first approach was given in [15] and for the second approach in [20]. The lower bound appears in [15].

Theorem 23 ([15, 20]) *Let* V *be a set of local variables and* P *be a set of atomic propositions. The satisfiability and model-checking problems for formulas of LTL extended with suffix implication, local variables, and the intersection operator using the semantics of* [15] *or* [20] *are EXPSPACE-complete with respect to* $|V| \cdot |P|$.

The source of the increase in complexity lies in the need to track different values for the same local variable on the same time instance of the same run. This issue was analyzed in [5]. It was shown there that in a significant syntactic fragment of PSL this situation will not occur. Given an alternating Büchi automaton for the formula at hand, roughly speaking, we may have more than one value for a certain local variable on a certain time point of the same state (a *local variable conflict*) if there are assignments after a universal branch that loops back. It was shown in [5] that on the subset given in Fig. 17, referred to as the *practical subset*, there will be no conflicts, and the complexity of model checking goes back to PSPACE.

Theorem 24 ([5]) *The space complexity of the verification problem of any formula* φ *in* PSL^{pract} *is polynomial in* $|\varphi|$.

Let b be a Boolean expression. Let Y be a sequence of local variables and E a sequence of expressions of the same length as Y. The grammar below defines the formulae φ that compose the practical subset, denoted PSL^{pract}, where the R operator is a dual to strong until: $\varphi \mathrm{R} \psi \equiv \neg[\neg\varphi \mathrm{U} \neg\psi]$.

$$r ::= b \mid r \cdot r \mid r \cup r \mid r^{+}$$

$$\mathcal{R} ::= b \mid (b, \mathrm{Y} := \mathrm{E}) \mid \mathcal{R} \cdot \mathcal{R} \mid \mathcal{R} \cup \mathcal{R} \mid \mathcal{R}^{+} \mid (\textsc{new}(\mathrm{Y})\ \mathcal{R}) \mid (\textsc{free}(\mathrm{Y})\ \mathcal{R})$$

$$\psi ::= r! \mid \neg r! \mid \psi \vee \psi \mid \psi \wedge \psi \mid \mathrm{X}!\psi \mid [\psi \mathrm{U} \psi] \mid [\psi \mathrm{R} \psi] \mid r \Longmapsto \psi$$

$$\varphi ::= \neg\mathcal{R}! \mid \psi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \mathrm{X}!\varphi \mid [\varphi \mathrm{U} \psi] \mid [\psi \mathrm{R} \varphi] \mid \mathcal{R} \Longmapsto \psi \mid (\textsc{new}(\mathrm{Y})\ \varphi) \mid (\textsc{free}(\mathrm{Y})\ \varphi)$$

Fig. 17 The practical subset, PSL^{pract} [5]

24.7 Summary and Open Issues

We have seen that both PSL and SVA extend the expressive power of LTL to that of ω-regular languages by adding regular expressions and the *suffix implication* operator. They both also include a number of specialized operators to allow natural specification of sampling abstractions and truncated paths. Finally, both provide local variables, which can be seen as a mechanism for both declaring quantified variables and constraining their behavior. But neither PSL nor SVA is perfect, nor necessarily done. Below we mention a few directions in which we think there might be room for improvement.

24.7.1 One-to-One Correspondence

The User's Point of View Consider the property that every request must receive a unique grant. This property is a common one, and most hardware designers need to express something of this sort.

Semantic Issues It is well known that the property $\{w \mid |w|_a = |w|_b\}$, where $|w|_\ell$ denotes the number of occurrences of the letter ℓ in word w, is not regular (see for example [42]), and cannot be expressed in any of the logics that we have considered in this chapter. On the one hand, this seems to be a non-issue. In every piece of hardware that obeys the desired property, there can be only a finite number n of requests outstanding (i.e., that have not yet received a grant), and Formulas (35)–(38) state the property for particular n's. Also, if the requests are tagged with unique identifiers, then the problem goes away.

However, from the user's point of view, the expressive power is not enough. She does not want a formula for a specific n, but rather one stating that there exists such an n. Therein lies an interesting paradox. Although ω-regular expressive

power is sufficient to describe the behavior of any particular design, the wish list of the user includes more. This is because we often want a specification to describe not a particular implementation, but a family of implementations, for instance all implementations in which every request receives a unique grant.

24.7.2 Triggering Procedural Code from Within a Formula

The User's Point of View Temporal logic is such a powerful tool, wouldn't it be nice to harness that power in order to "trigger" procedural code? A simple motivation for this would be to print out debugging information when the calculation "reaches" a particular point in the formula. Another application would be to drive simulation inputs or to collect coverage information.

Such an ability exists in SVA [45, 46], in which a subroutine call is allowed wherever a local variable assignment is allowed, and in PSL 1850–2010 [44], where a *procedural block* is similarly allowed.

Semantic Issues Triggering of procedural code from within a formula can be viewed as an extension of local variables as discussed in Sect. 24.6. However, whereas the semantics of local variables are well defined, and given by the formal semantics of SVA and PSL, the semantics of triggering procedural code are not formally defined at all in either specification language.

Doing so is non-trivial, not only because it is difficult to write down the precise, mathematically well-behaved semantics, but also because first it must be decided what they are. For example, what side effects are allowed in the procedural code (i.e., how does the procedural code interact with the design under verification)? And if a property consists of the disjunction of two sub-properties, one of which has already been determined to hold, must we continue to trigger procedural code attached to the other sub-property if it holds as well? And if a regular expression is "matched" in two different ways at the same cycle, do we trigger the relevant procedural code once or twice? Note that any decision on such issues limits the implementation of a tool checking properties in the specification language.

24.7.3 Separation of Concerns

The User's Point of View Consider the property "If p occurs followed eventually by q, then v will not occur between p and q, will not occur at q, and furthermore will occur one cycle after q." This can be expressed in LTL as follows:

$$\mathrm{G}\big(p \rightarrow \big(\big[\neg v \ \mathrm{W}\ (q \wedge \neg v)\big] \wedge \big[\neg q \ \mathrm{W}\ (q \wedge \mathrm{X} v)\big]\big)\big) \tag{41}$$

However, this formulation blurs the distinction between cause and effect: in the English language description of the formula it is quite clear that the "responsibility"

for the property lies with signal v, but in an LTL formula all variables are created equal, so to speak.

Semantic Issues The hardware specification language ITL [73] (distinct from the logic of the same name described in [36]) enforces a "separation of concerns," by providing a syntactic distinction between the antecedent and the consequent of every formula. For this example, separating cause from effect clearly is possible in PSL and SVA as well. For example, the following pair of formulas is together equivalent to Formula (41) while restricting the p's and q's to be on the left and the v's to be on the right of a suffix implication operator:

$$\mathrm{G}\big(\big(p \cdot \neg q^*\big) \mapsto \neg v\big) \tag{42}$$

$$\mathrm{G}\big(\big(p \cdot \neg q^* \cdot q\big) \mapsto (\neg v \cdot v)\big) \tag{43}$$

However, this approach is quite far from the philosophy of ITL, in which the separation of cause and effect is enforced by the syntax of the language. It would be interesting to see whether there is an elegant way to incorporate the separation of concerns into PSL and SVA without breaking useful features of these languages.

Acknowledgements Thank you to Gadi Aleksandrowicz, Shoham Ben-David, Alexander Ivrii, Avigail Orni, Sitvanit Ruah, Moshe Vardi, and anonymous reviewers of this chapter for useful comments.

References

1. Abarbanel, Y., Beer, I., Gluhovsky, L., Keidar, S., Wolfsthal, Y.: FoCs: automatic generation of simulation checkers from formal specifications. In: Emerson, E.A., Sistla, A.P. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 1855, pp. 538–542. Springer, Heidelberg (2000)
2. Alpern, B., Schneider, F.B.: Defining liveness. Inf. Process. Lett. **21**(4), 181–185 (1985)
3. Alpern, B., Schneider, F.B.: Recognizing safety and liveness. Distrib. Comput. **2**(3), 117–126 (1987)
4. Armoni, R., Bustan, D., Kupferman, O., Vardi, M.: Resets vs. aborts in linear temporal logic. In: Garavel, H., Hatcliff, J. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 2619, pp. 65–80. Springer, Heidelberg (2003)
5. Armoni, R., Fisman, D., Jin, N.: SVA and PSL local variables—a practical approach. In: Sharygina, N., Veith, H. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 8044, pp. 197–212. Springer, Heidelberg (2013)
6. Armoni, R., Fix, L., Flaisher, A., Gerth, R., Ginsburg, B., Kanza, T., Landver, A., Mador-Haim, S., Singerman, E., Tiemeyer, A., Vardi, M., Zbar, Y.: The ForSpec temporal logic: a new temporal property-specification language. In: Katoen, J., Stevens, P. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 2280, pp. 296–311. Springer, Heidelberg (2002)
7. Beer, I., Ben-David, S., Eisner, C., Fisman, D., Gringauze, A., Rodeh, Y.: The temporal logic Sugar. In: Berry, G., Comon, H., Finkel, A. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 2102, pp. 363–367. Springer, Heidelberg (2001)

8. Beer, I., Ben-David, S., Landver, A.: On-the-fly model checking of RCTL formulas. In: Hu, A.J., Vardi, M.Y. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 1427, pp. 184–194. Springer, Heidelberg (1998)
9. Ben-David, S., Bloem, R., Fisman, D., Griesmayer, A., Pill, I., Ruah, S.: Automata construction algorithms optimized for PSL (Deliverable 3.2/4). Tech. rep., PROSYD (2005)
10. Ben-David, S., Fisman, D., Ruah, S.: The safety simple subset. In: Ur, S., Bin, E., Wolfsthal, Y. (eds.) Intl. Haifa Verification Conference (HVC). LNCS, vol. 3875, pp. 14–29. Springer, Heidelberg (2005)
11. Ben-David, S., Fisman, D., Ruah, S.: Embedding finite automata within regular expressions. Theor. Comput. Sci. **404**(3), 202–218 (2008)
12. Bloem, R., Ravi, K., Somenzi, F.: Efficient decision procedures for model checking of linear time logic properties. In: Halbwachs, N., Peled, D. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 1633, pp. 222–235. Springer, Heidelberg (1999)
13. Bustan, D., Fisman, D., Havlicek, J.: Automata construction for PSL. Tech. Rep. MCS05-04, The Weizmann Institute of Science (2005)
14. Bustan, D., Flaisher, A., Grumberg, O., Kupferman, O., Vardi, M.: Regular vacuity. In: Correct Hardware Design and Verification Methods (CHARME). LNCS, vol. 3725, pp. 191–206. Springer, Heidelberg (2005)
15. Bustan, D., Havlicek, J.: Some complexity results for SystemVerilog assertions. In: Ball, T., Jones, R.B. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 4144, pp. 205–218. Springer, Heidelberg (2006)
16. Cerny, E., Dudani, S., Havlicek, J., Korchemny, D.: The Power of Assertions in SystemVerilog. Springer, Heidelberg (2010)
17. Clarke, E., Emerson, E.: Design and synthesis of synchronization skeletons using branching-time temporal logic. In: Kozen, D. (ed.) Workshop on Logic of Programs. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1981)
18. Eisner, C., Fisman, D.: Sugar 2.0 proposal presented to the Accellera Formal Verification Technical Committee (2002). At http://www.haifa.il.ibm.com/projects/verification/sugar/Sugar_2.0_Accellera.ps
19. Eisner, C., Fisman, D.: A Practical Introduction to PSL. Springer, New York (2006)
20. Eisner, C., Fisman, D.: Augmenting a regular expression-based temporal logic with local variables. In: Cimatti, A., Jones, R.B. (eds.) Formal Methods in Computer Aided Design (FMCAD), pp. 1–8. IEEE, Piscataway (2008)
21. Eisner, C., Fisman, D.: Structural contradictions. In: Chockler, H., Hu, A.J. (eds.) Intl. Haifa Verification Conference (HVC). LNCS, vol. 5394, pp. 164–178. Springer, Heidelberg (2008)
22. Eisner, C., Fisman, D., Havlicek, J.: A topological characterization of weakness. In: Aguilera, M.K., Aspnes, J. (eds.) Symp. on Principles of Distributed Computing (PODC), pp. 1–8. ACM, New York (2005)
23. Eisner, C., Fisman, D., Havlicek, J.: Safety and liveness, weakness and strength, and the underlying topological relations. ACM Trans. Comput. Log. **15**(2), 13:1–13:44 (2014)
24. Eisner, C., Fisman, D., Havlicek, J., Lustig, Y., McIsaac, A., Van Campenhout, D.: Reasoning with temporal logic on truncated paths. In: Hunt, W.A. Jr., Somenzi, F. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 2725, pp. 27–39. Springer, Heidelberg (2003)
25. Eisner, C., Fisman, D., Havlicek, J., Mårtensson, J.: The $\top, \bot$ approach for truncated semantics. Tech. Rep. 2006.01, Accellera (2006)
26. Eisner, C., Fisman, D., Havlicek, J., McIsaac, A., Van Campenhout, D.: The definition of a temporal clock operator. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) Intl. Coll. on Automata, Languages and Programming (ICALP). LNCS, vol. 2719, pp. 857–870. Springer, Heidelberg (2003)
27. Emerson, E.: Temporal and modal logic. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, Volume B, pp. 995–1072. Elsevier/MIT Press, Amsterdam/Cambridge (1990). Chap. 16

28. Fischer, M., Ladner, R.: Propositional dynamic logic of regular programs. J. Comput. Syst. Sci. **18**, 194–211 (1979)
29. Fisman, D.: On the characterization of until as a fixed point under clocked semantics. In: Yorav, K. (ed.) Intl. Haifa Verification Conference (HVC). LNCS, vol. 4899, pp. 19–33. Springer, Heidelberg (2007)
30. French, T., Reynolds, M.: A sound and complete proof system for QPTL. In: Balbiani, P., Suzuki, N., Wolter, F., Zakharyaschev, M. (eds.) Advances in Modal Logic, pp. 127–148. King's College Publications, London (2002)
31. Gabbay, D., Pnueli, A., Shelah, S., Stavi, J.: On the temporal analysis of fairness. In: Abrahams, P.W., Lipton, R.J., Bourne, S.R. (eds.) Symp. on Principles of Programming Languages (POPL), pp. 163–173. ACM, New York (1980)
32. Gelade, W.: Succinctness of regular expressions with interleaving, intersection and counting. In: Ochmanski, E., Tyszkiewicz, J. (eds.) Intl. Symp. on Mathematical Foundations of Computer Science (MFCS). LNCS, vol. 5162. Springer, Heidelberg (2008)
33. Gelade, W., Neven, F.: Succinctness of the complement and intersection of regular expressions. In: Annual Symp. on Theoretical Aspects of Computer Science (STACS). Leibniz International Proceedings in Informatics, vol. 1, pp. 325–336. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl (2008)
34. Gordon, M.: Validating the PSL/Sugar semantics using automated reasoning. Form. Asp. Comput. **15**(4), 406–421 (2003)
35. Hafer, T., Thomas, W.: Computation tree logic CTL* and path quantifiers in the monadic theory of the binary tree. In: Ottmann, T. (ed.) Intl. Coll. on Automata, Languages and Programming (ICALP). LNCS, vol. 267, pp. 269–279. Springer, Heidelberg (1987)
36. Halpern, J.Y., Manna, Z., Moszkowski, B.C.: A hardware semantics based on temporal intervals. In: Díaz, J. (ed.) Intl. Coll. on Automata, Languages and Programming (ICALP). LNCS, vol. 154, pp. 278–291. Springer, Heidelberg (1983)
37. Harel, D., Kozen, D., Parikh, R.: Process logic: expressiveness, decidability, completeness. J. Comput. Syst. Sci. **25**(2), 144–170 (1982)
38. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. MIT Press, Cambridge (2000)
39. Harel, D., Peleg, D.: Process logic with regular formulas. Theor. Comput. Sci. **38**, 307–322 (1985)
40. Havlicek, J., Shultz, K., Armoni, R., Dudani, S., Cerny, E.: Notes on the semantics of local variables in Accellera SystemVerilog 3.1 concurrent assertions. Tech. Rep. 2004.01, Accellera (2004)
41. Holzer, M., Jakobi, S.: State complexity of chop operations on unary and finite languages. In: Kutrib, M., Moreira, N., Reis, R. (eds.) Workshop on Descriptional Complexity of Formal Systems (DCFS). LNCS, vol. 7386, pp. 169–182. Springer, Heidelberg (2012)
42. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley, Reading (1979)
43. IEEE Standard for Property Specification Language (PSL), Annex B. IEEE Std 1850™-2005
44. IEEE Standard for Property Specification Language (PSL), Annex B. IEEE Std 1850™-2010
45. IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language, Annex E. IEEE Std 1800™-2005
46. IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language, Annex F. IEEE Std 1800™-2009
47. Kamp, J.: Tense logic and the theory of order. Ph.D. thesis, Univ. of California, Los Angeles (1968)
48. Kesten, Y., Pnueli, A.: A complete proof system for QPTL. In: Proceedings, Tenth Annual IEEE Symposium on Logic in Computer Science, pp. 2–12. IEEE, Piscataway (1995)

49. Kilpeläinen, P., Tuhkanen, R.: Regular expressions with numerical occurrence indicators—preliminary results. In: Kilpeläinen, P., Päivinen, N. (eds.) Symp. on Programming Languages and Software Tools (SPLST), pp. 163–173 (2003). University of Kuopio, Department of Computer Science
50. Kupferman, O., Vardi, M.Y.: Model checking of safety properties. In: Halbwachs, N., Peled, D. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 1633, pp. 172–183. Springer, Heidelberg (1999)
51. Lange, M.: Linear time logics around PSL: complexity, expressiveness, and a little bit of succinctness. In: Caires, L., Vasconcelos, V.T. (eds.) Intl. Conf. on Concurrency Theory (CONCUR). LNCS, vol. 4703, pp. 90–104. Springer, Heidelberg (2007)
52. Laroussinie, F., Markey, N., Schnoebelen, P.: Temporal logic with forgettable past. In: Kohlenbach, U., Abramsky, S. (eds.) Symp. on Logic in Computer Science (LICS), pp. 383–392. IEEE, Piscataway (2002)
53. Lichtenstein, O., Pnueli, A.: Checking that finite state concurrent programs satisfy their linear specification. In: Deusen, M.S.V., Galil, Z., Reid, B.K. (eds.) Symp. on Principles of Programming Languages (POPL), pp. 97–107. ACM, New York (1985)
54. Lichtenstein, O., Pnueli, A., Zuck, L.D.: The glory of the past. In: Parikh, R. (ed.) Logic of Programs. LNCS, vol. 193, pp. 196–218. Springer, Heidelberg (1985)
55. Long, J., Seawright, A., Kavalipati, P.: Multi-clock SVA synthesis without re-writing. In: Wakabayashi, K. (ed.) Asia and South Pacific Design Automation Conf. (ASPDAC), pp. 648–653. IEEE, Piscataway (2009)
56. Maidl, M.: The common fragment of CTL and LTL. In: Annual Symp. on Foundations of Computer Science (FOCS), pp. 643–652. IEEE, Piscataway (2000)
57. McMillan, K.: Symbolic Model Checking. Kluwer Academic, Norwell (1993)
58. McNaughton, R., Papert, S.A.: Counter-Free Automata (MIT Research Monograph No. 65). MIT Press, Cambridge (1971)
59. Morley, M.: Semantics of temporal e. In: Melham, T.F., Moller, F.G. (eds.) Proc. Banff Higher Order Workshop. Formal Methods in Computation (1999). Univ. of Glasgow, Dept. of Computing Science. Technical Report
60. Muller, D.E., Saoudi, A., Schupp, P.E.: Weak alternating automata give a simple explanation of why most temporal and dynamic logics are decidable in exponential time. In: Symp. on Logic in Computer Science (LICS), pp. 422–427. IEEE, Piscataway (1988)
61. Oliveira, M.T., Hu, A.J.: High-level specification and automatic generation of IP interface monitors. In: Design Automation Conf. (DAC), pp. 129–134. ACM, New York (2002)
62. Pnueli, A.: The temporal logic of programs. In: Annual Symp. on Foundations of Computer Science (FOCS), pp. 46–57. IEEE, Piscataway (1977)
63. Pratt, V.R.: Semantical considerations on Floyd-Hoare logic. Tech. rep., Massachusetts Inst. of Technology (1976)
64. Seawright, A., Brewer, F.: High-level symbolic construction technique for high performance sequential synthesis. In: Dunlop, A.E. (ed.) Design Automation Conf. (DAC), pp. 424–428. IEEE, Piscataway (1993)
65. Sistla, A.P.: Theoretical issues in the design and verification of distributed systems. Ph.D. thesis, Harvard Univ. (1983)
66. Sistla, A.P., Clarke, E.M.: The complexity of propositional linear temporal logics. J. ACM **32**(3), 733–749 (1985)
67. Sistla, A.P., Vardi, M.Y., Wolper, P.L.: The complementation problem for Büchi automata, with applications to temporal logic. Theor. Comput. Sci. **49**, 217–237 (1987)
68. Thomas, W.: Star-free regular sets of ω-sequences. Inf. Control **42**(2), 148–156 (1979)
69. Vardi, M.: Branching vs. linear time: final showdown. In: Margaria, T., Yi, W. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 2031, pp. 1–22. Springer, Heidelberg (2001)
70. Vardi, M.Y.: From Church and Prior to PSL. In: Grumberg, O., Veith, H. (eds.) 25 Years of Model Checking. LNCS, vol. 5000, pp. 150–171. Springer, Heidelberg (2008)

71. Vardi, M.Y., Wolper, P.: Yet another process logic (preliminary version). In: Clarke, E.M., Kozen, D. (eds.) Workshop on Logic of Programs. LNCS, vol. 164, pp. 501–512. Springer, Heidelberg (1983)
72. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification (preliminary report). In: Symp. on Logic in Computer Science (LICS), pp. 332–344. IEEE, Piscataway (1986)
73. Winkelmann, K., Trylus, H., Stoffel, D., Fey, G.: Cost-efficient block verification for a UMTS up-link chip-rate coprocessor. In: Design, Automation & Test in Europe (DATE), pp. 162–167. IEEE, Piscataway (2004)
74. Wolper, P.: Temporal logic can be more expressive. Inf. Control **56**(1/2), 72–99 (1983)

Chapter 25
Symbolic Trajectory Evaluation

Tom Melham

Abstract Symbolic trajectory evaluation is an industrial-strength formal hardware verification method, based on symbolic simulation, which has been highly successful in data-path verification, especially for microprocessor execution units. It is a 'model-checking' method in the basic sense that properties, expressed in a simple temporal logic, are verified by (symbolic) exploration of formal models of sequential circuits. Its defining characteristic is that it operates by symbolic simulation over *abstractions* of sets of states that only partially delineate the circuit states in the set. These abstract state sets are ordered in a lattice by information content, based on a three-valued domain for values on circuit nodes (*true*, *false*, and *don't know*). The algorithm operates over families of these abstractions encoded by Boolean formulas, providing a flexible, specification-driven mechanism for partitioned data abstraction. We provide a basic introduction to symbolic trajectory evaluation and its extensions, and some details of how it is deployed in industrial practice. The aim is to get across the essence and value of the method in clear and accessible terms.

25.1 Introduction

Symbolic Trajectory Evaluation (STE) is a model-checking method based on symbolic simulation over a lattice of abstract state sets [64]. STE's combination of abstraction and algorithmic efficiency is especially suited to verification of large datapaths and memories, and has been demonstrated on many hard industrial verification problems at Intel Corporation [42, 52, 58]. Two notable successes are the verification, using Intel's Forte system [65], of the entire execution cluster of the Intel Core 2 Duo and Core i7 processors [26, 41]. Motorola has also used STE extensively for verification of embedded custom memories [45], and has developed a suite of advanced methodologies and tools to support this task [10, 11, 71].

In the abstraction lattice at the heart of STE, each circuit node (i.e. wire carrying a single bit) is assigned a value in the set $\{X, 0, 1\}$, with 'X' representing an 'unknown' value. An assignment of such values to every circuit node is an abstraction

T. Melham (✉)
University of Oxford, Oxford, UK
e-mail: tom.melham@cs.ox.ac.uk

E.M. Clarke et al. (eds.), *Handbook of Model Checking*,
DOI 10.1007/978-3-319-10575-8_25

of a set of Boolean circuit states. It is abstract in the sense that it ambiguously stands for any one of a family of Boolean state sets, one for each replacement of every X by 0 or 1. The collection of all such abstractions forms a lattice, ordered by the amount of information we have about node values.

The STE model-checking algorithm uses three-valued circuit simulation [14] to compute a reachable abstract state-set in this representation. The algorithm is space-efficient because it operates over abstractions of sets of states; any parts of the circuit function not relevant to the specification get 'abstracted away' to X. Any correctness result verified in this abstract model transfers over to the real, Boolean model of circuit states. Formally, there is a Galois connection between the three-valued model and the Boolean model of states [19].

This abstraction machinery is controlled by the way in which the user writes properties for model checking. By careful coding of the property, the user can guide the symbolic simulation done during model checking through the right layers of the abstract state lattice to verify the property with contained complexity. A good illustration of success is the content-addressable memory verification done by Pandey and colleagues [52], in which a careful encoding of properties gives a logarithmic reduction in complexity.

Specifications themselves are written in a very simple linear-time temporal logic, limited to implications between formulas with only conjunction and the next-time operator. STE specifications therefore express only bounded time properties, and it may take more properties to verify the same functionality in STE than in, say, CTL—the approach is in some ways similar to the FSM decomposition methodology of Interval Property Checking [50]. Different versions and extensions of STE with more expressive logics exist [32, 35, 64, 77]. But the simple form is the most widely tested on industrial applications and the focus of this chapter.

Although the logic of STE seems weak, its expressive power and abstraction capability are greatly enhanced by use of *symbolic* ternary simulation [16]. On top of the abstraction lattice, STE provides a layer of symbolic representation whereby whole *families* of abstractions, each covering only a part of the circuit's function, may be checked simultaneously. This mechanism, sometimes called 'symbolic indexing', provides a flexible way to achieve case-splitting by a data abstraction scheme. A characteristic example is a memory verification, in which an n-element memory is verified with an indexed family of n abstractions, one for each address at which some target data might be located.

In STE, the cases covered by the abstractions in an indexed family can overlap, in contrast to existential abstraction for CTL [22, 28], which partitions the state space. Symbolic indexing can also record inter-dependencies among node values, and so increases the expressive power of specifications for STE. For example, input/output functions can be extracted from a circuit by using symbolic simulation to derive formulas for the values on output nodes as functions of variables standing for arbitrary values on input nodes. These can then be checked against a specification in the form of some reference formulas provided by the verification engineer. Disjunction can also be expressed.

This chapter explains the theory of STE model checking in detail and briefly describes how the method is implemented and used in practice. It also gives a brief

sketch of some extensions to STE, including Generalized Symbolic Trajectory Evaluation (GSTE). A full account of the theory of STE by its originators, Carl Seger and Randy Bryant, can be found in [64]. An illuminating perspective on the foundations of STE is provided by Chou in [19]. There is an in-depth tutorial on STE by Seger and Hazelhurst [32]. Roorda and Claessen also provide a tutorial account of STE in [20] and discuss its semantics in [54]. A usage methodology for STE in industrial practice can be found in [1, 38, 65]. GSTE is introduced and explained in [21, 67, 68, 77], and its relationship to conventional symbolic model checking is explored in [59].

25.2 Notational Preliminaries

We write $\stackrel{\triangle}{=}$ to mean *equals by definition*. We assume familiarity with elementary propositional logic and predicate calculus notation and use the symbol $\supset$ for logical implication. We denote the set of Boolean truth-values by $\mathbb{B} = \{\mathrm{T}, \mathrm{F}\}$. We use lower-case letters (e.g. a, v, x, y_1) as Boolean variables, and use upper-case letters (e.g. P, Q) to stand for formulas of propositional logic ('Boolean functions'). We write $\vec{x}$ to mean a vector of unique variables $x_1, \ldots, x_n$ for indeterminate n and similarly $\vec{P}$ to stand for a vector of formulas $P_1, \ldots, P_n$.

The notation $P[\vec{Q}/\vec{x}]$ stands for the result of simultaneously substituting the formulas $\vec{Q}$ for all occurrences of the respective Boolean variables $\vec{x}$ in P. The notation $P[\vec{x}]$ should be taken to mean a formula that may contain free occurrences of the distinct Boolean variables $\vec{x}$. In a context in which a formula has been written $P[\vec{x}]$, subsequent use of the notation $P[\vec{Q}]$ can then be understood to mean the result of substituting the formulas $\vec{Q}$ for the variables $\vec{x}$ in $P[\vec{x}]$.

If P is a propositional formula and ϕ is a function that assigns a truth-value to each Boolean variable in it, we write $\phi \Vdash P$ to mean that ϕ satisfies P [56].

We also assume familiarity with the notation of naive set theory [29] and the rudiments of conventional functional programming notation for higher-order functions [12]. If A and B are sets, we write $A \to B$ for the set of all total functions from A to B. We assume that $\to$ associates to the right, so $A \to (B \to C)$ may be written $A \to B \to C$. We write function applications by mere juxtaposition: $f\ x$ instead of $f(x)$. For higher-order functions, function application associates to the left; so if $f \in A \to B \to C$, $a \in A$, and $b \in B$, then we can write $f\ a\ b$ for $(f\ a)\ b$.

The semantics of symbolic trajectory evaluation uses some elementary concepts of lattice theory [24]. A *poset* $(S, \sqsubseteq)$ is a partial order $\sqsubseteq$ on a set S. If $(S, \sqsubseteq)$ is a poset and $A \subseteq S$, then $x \in S$ is an *upper bound* for A exactly when $a \sqsubseteq x$ for all $a \in A$. A *lower bound* is defined dually. An upper bound x of A is the *least upper bound* of A, written $\sqcup A$, if $x \sqsubseteq y$ for every upper bound y of A. The *greatest lower bound*, written $\sqcap A$, is defined dually. We also write $a \sqcup b$ (read 'a join b') for $\sqcup\{a, b\}$ when it exists and $a \sqcap b$ (read 'a meet b') for $\sqcap\{a, b\}$ when it exists.

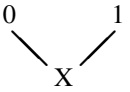
A poset $(S, \sqsubseteq)$ is a *complete lattice* iff $\sqcup A$ and $\sqcap A$ exist for all $A \subseteq S$. If S is finite and $a \sqcup b$ and $a \sqcap b$ exist for all $a, b \in S$, then $(S, \sqsubseteq)$ is a complete lattice.

25.3 Sequential Circuit Models in STE

We suppose there is a finite set of circuit nodes $\mathscr{N}$, naming observable points in circuits. We can think of a node as the name of a wire—i.e. just an identifier, such as 'reset' or 'input32'. As usual for sequential circuit models, there is a node in $\mathscr{N}$ to name each primary circuit input, as well as a node to name the output of each latch or other state-holding element. In STE, we can also give names to selected internal wires within a block of combinational logic, if we wish to express verification properties that make reference to values on these wires. In the interest of clarity, however, we will ignore this possibility for now.

25.3.1 States and Sequences

To describe the behaviour of a circuit formally, we need to say which succession of values will be present on each of its nodes as the circuit evolves over time. Symbolic trajectory evaluation employs a three-valued state model, with values drawn from the set $\mathscr{D} = \{\mathrm{X}, 0, 1\}$. The usual binary values 0 and 1 are augmented with an additional value X. This stands for an unknown, which we represent mathematically by a partial order $\leq$ on $\mathscr{D}$, in which $\mathrm{X} \leq 0$ and $\mathrm{X} \leq 1$:



This orders values by information content: X stands for an unknown value and so is ordered below 0 and 1.

A *state* is an instantaneous snapshot of circuit behaviour given by an assignment of values in $\mathscr{D}$ to all the node names in $\mathscr{N}$. A state is represented mathematically by a function

$$s \in \mathscr{N} \to \mathscr{D}$$

that maps each node name to a value.

If the set of circuit nodes is small enough, we can write down specific states just by giving the function explicitly. For example, for $\mathscr{N} = \{\mathsf{a},\ \mathsf{b},\ \mathsf{c}\}$, we might write

$$\big\{(\mathsf{a}, 0),\ (\mathsf{b}, \mathrm{X}), (\mathsf{c}, 1)\big\} \quad \text{or} \quad \{\mathsf{a} \mapsto 0,\ \mathsf{b} \mapsto \mathrm{X}, \mathsf{c} \mapsto 1\}$$

for the state in which a is 0, b is X, and c is 1. We can use a more compact notation if we give an ordering on nodes. If the nodes in this example are ordered $\mathsf{a} < \mathsf{b} < \mathsf{c}$, we can just write '0X1' to denote this state. In what follows, we will feel free to use either of these notations, as appropriate.

The ordering $\leq$ on $\mathscr{D}$ is extended point-wise to get an ordering $\sqsubseteq$ on states. For reasons that will be clear later, we wish this to form a complete lattice, and so we will use a special symbol, $\top$, for the top element of this state lattice. We then define

the set of states $\mathscr{S}$ to be $(\mathscr{N} \to \mathscr{D}) \cup \{\top\}$. The required ordering is defined for states $s_1, s_2 \in \mathscr{S}$ as follows:

$$s_1 \sqsubseteq s_2 \triangleq \begin{cases} s_2 = \top, \text{ or} \\ s_1, s_2 \in \mathscr{N} \to \mathscr{D} \text{ and } s_1(n) \leq s_2(n) \text{ for all } n \in \mathscr{N} \end{cases}$$

The intuition is that if $s_1 \sqsubseteq s_2$, then s_1 may have 'less information' about node values than s_2—i.e. it may have Xs in place of certain 0s and 1s.

25.3.2 Abstraction

The term *state* just introduced is really a misnomer: a 'state' in STE, such as 0X1, is really an abstraction—an *abstract predicate* on sets of actual circuit states. Real circuit states are always Boolean-valued: each circuit node is either 0 or 1 and there isn't really a *value* 'X'. Roughly speaking, we can think of a lattice state as standing for a set of ordinary, Boolean-valued states. More precisely, a single lattice state stands (ambiguously, because it is an abstraction) for any one of a certain group of structurally related sets of Boolean states.

Computational manipulation of sets of circuit states is a key idea in classical model checking. STE model checking also operates over sets of Boolean circuit states, but there is an abstraction: the sets are only incompletely known or specified. This abstraction scheme is related to the *Cartesian abstraction* employed in some approaches to software model checking [6].

Suppose we have only two circuit nodes, a and b, and for the purpose of compactly writing down lattice states, we order them $\mathsf{a} < \mathsf{b}$. The lattice state 0X can be seen as an abstraction of the set of Boolean-valued states

$$\big\{\{\mathsf{a} \mapsto 0,\ \mathsf{b} \mapsto 0\},\ \{\mathsf{a} \mapsto 0,\ \mathsf{b} \mapsto 1\}\big\}.$$

In the abstract 'state' 0X the node b is assigned the unknown value X. In the concrete set of Boolean states, there is one state in which b is 0 and one in which b is 1. So the node b can have either value. In STE, the lattice state 0X is also an abstraction of the following singleton set of Boolean-valued states:

$$\big\{\{\mathsf{a} \mapsto 0,\ \mathsf{b} \mapsto 0\}\big\}.$$

Here, node b must be 0. Finally, the lattice state 0X is also an abstraction of the set of Boolean states

$$\big\{\{\mathsf{a} \mapsto 0,\ \mathsf{b} \mapsto 1\}\big\},$$

in which node b must be 1. The lattice-valued 'state' 0X is an abstraction of all three of these sets of real circuit states. That is, there is loss of information, the defining characteristic of being an 'abstraction'.

In practice, we can often think informally of a lattice state s as standing for the set of Boolean states obtainable by replacing all occurrences of X in s by a combination

of 0s and 1s in every possible way. Every set of Boolean states of which s is an abstraction is some subset of this 'maximal' set of Boolean states. A characteristic of this abstraction is that it ignores information about dependencies between node values. Consider, for example, the set of states $\{\{\mathsf{a} \mapsto 0,\ \mathsf{b} \mapsto 1\}, \{\mathsf{a} \mapsto 1,\ \mathsf{b} \mapsto 0\}\}$, in which a and b always have values that are the negation of each other. The only abstraction of this set of states is XX.

One can view abstract 'states' s_1 and s_2 as *constraints* or *predicates* on sets of actual states of the hardware. If $s_1 \sqsubseteq s_2$, then every set of Boolean states that satisfies s_2 also satisfies s_1. For example, suppose we again have nodes $\mathsf{a} < \mathsf{b}$ and consider $0\mathrm{X} \sqsubseteq 01$. Lattice state 01 unambiguously stands for the singleton set of Boolean states $\{\{\mathsf{a} \mapsto 0,\ \mathsf{b} \mapsto 1\}\}$, which also satisfies the 'constraint' 0X. More precisely, any set of states of which s_2 is an abstraction is a subset of any set of states of which s_1 is an abstraction. We say that s_1 is 'weaker than' s_2. (Strictly speaking, $\sqsubseteq$ is reflexive and we really mean 'no stronger than', but it is common to be a little inexact and just say 'weaker than'.) The top value $\top$ represents the unsatisfiable constraint. The *join* operator on pairs of states in the lattice is denoted by '$\sqcup$'.

25.3.3 Time-Dependent Behaviour

To model dynamic behaviour, we represent time by the natural numbers $\mathbb{N}$. A sequence of states that the circuit passes through as it evolves over time is then represented by a function from time to states:

$$\sigma \in \mathbb{N} \to \mathscr{S}.$$

Such a function is called a *sequence*. A sequence (that never produces the over-constrained top state $\top$) just assigns a value in $\{\mathrm{X}, 0, 1\}$ to each circuit node at each point in time. For example, $\sigma\ 3\ \mathsf{reset}$ would be the value assigned to the reset node at time 3.

The ordering on states is extended point-wise to sequences in the usual way:

$$\sigma_1 \sqsubseteq \sigma_2 \stackrel{\triangle}{=} \sigma_1(t) \sqsubseteq \sigma_2(t) \quad \text{for all } t \in \mathbb{N}.$$

If $\sigma_1 \sqsubseteq \sigma_2$, then we say that the sequence σ_1 is 'weaker than' the sequence σ_2. As before, it would be more accurate to say 'no stronger than'.

We now introduce an operation on sequences that is used later in stating the semantics of STE. For any $i \geq 0$, the ith *suffix* of a sequence σ is written σ^i and defined by

$$\sigma^i\ t \stackrel{\triangle}{=} \sigma\ (t+i) \quad \text{for all } t \in \mathbb{N}.$$

Taking the ith suffix just shifts the sequence σ forward i points in time, discarding the first i states.

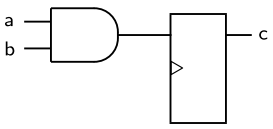


Fig. 1 Unit-delay AND-gate

25.3.4 The Next State Function and Trajectories

In symbolic trajectory evaluation, the formal model of a circuit c is given by a next-state function Y_c that maps states to states:

$$\mathrm{Y}_c \in \mathscr{S} \to \mathscr{S}.$$

The subscript in Y_c identifies the particular circuit of interest; we can think of 'c' as a constant that names the circuit. This subscript is frequently omitted, as it is usually clear or doesn't matter which circuit is being discussed.

Intuitively, the next-state function expresses a constraint on the set of possible Boolean states into which the circuit may go for any set of Boolean states it might currently be in. Suppose the circuit is in abstract state $s \in \mathscr{S}$. Then $\mathrm{Y}\, s$ is the most-specified abstract state that covers all the sets of states the circuit can make a transition to from any set of states that satisfies s. Roughly speaking, if the only possible value for a circuit node n in every next state is one of 0 or 1, then $\mathrm{Y}\, s$ will assign that value to n. Otherwise n will be X in the next abstract state.

A small example is the trivial unit-delay AND-gate in Fig. 1. This circuit has three nodes, which for the purpose of writing down states we order $\mathsf{a} < \mathsf{b} < \mathsf{c}$. Suppose the current state is 111, i.e. all nodes are known to be high. Then the next state Y(111) will be XX1. Similarly, if the current state is 110, then the next state Y(110) will also be XX1. In fact, the next value of c doesn't depend on its value in the current state, so $\mathrm{Y}(11\mathrm{X}) = \mathrm{XX1}$ as well. In all these cases, we see that in the next state a and b are both X because they are primary inputs—they are 'non-deterministic'. If b is 0 in the current state, then c is going to be 0 in the next state, regardless of the value of a in the current state. Hence $\mathrm{Y}(\mathrm{X0X}) = \mathrm{XX0}$. Finally, we sometimes have insufficient information to determine the value of the output. If the current state is X1X, for example, then we don't know whether c is going to be 0 or 1. It may be either, and hence $\mathrm{Y}(\mathrm{X1X}) = \mathrm{XXX}$.

In essence, X serves as a 'don't care' (or, depending on context, 'don't know') value in STE reachable-state calculations. This allows an STE model checker to prune away parts of a circuit function that have no effect on the properties being validated—a kind of semantic 'cone of influence' reduction [23]. On the other hand, if not enough is known about initial states of the system, Xs can permeate the computations enough to make the satisfaction of target properties indeterminate. We discuss this issue later.

A requirement for trajectory evaluation is that the next-state function Y is monotonic. That is, for all states s_1 and s_2

$$s_1 \sqsubseteq s_2 \quad \text{implies} \quad \mathrm{Y}\, s_1 \sqsubseteq \mathrm{Y}\, s_2.$$

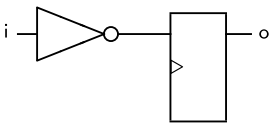


Fig. 2 Unit-delay inverter

This is consistent with the idea that $\sqsubseteq$ is an information ordering: any increase in knowledge about the range of possible current states should produce only an increase—not a reduction or revision—of knowledge about the range of possible next states. Implementations of STE are designed to extract a next-state function that has this property from the circuit under analysis. This condition can be met, with some careful engineering, for a wide variety of common circuit design styles, including synchronous systems with latches as well as flip-flops, and systems with gated clocks. Transistor switch-level models can also be devised for STE [32, 64].

For a circuit c, we define the set of all its *trajectories*, $\mathscr{T}_c$, as follows:

$$\mathscr{T}_c \triangleq \left\{\sigma \;\middle|\; \mathrm{Y}_c(\sigma\, t) \sqsubseteq \sigma\, (t{+}1) \text{ for all } t \in \mathbb{N}\right\}.$$

For a sequence σ to be a trajectory, each successor state in the sequence must be at least as specified as (and not contradict) the result of applying the next-state function Y_c to the previous state in the sequence. This ensures that σ is consistent with the circuit model Y_c, i.e. that the succession of abstract state constraints given by σ does not contradict what the hardware would actually do.

For example, suppose $\mathscr{N} = \{\mathsf{i}, \mathsf{o}\}$ with $\mathsf{i} < \mathsf{o}$ and consider the circuit in Fig. 2. The partial sequence $\sigma = \mathrm{1X}, \mathrm{XX}, \ldots$ does not begin a trajectory. In the second state of circuit execution we 'know' that o must be 0, because $\mathrm{Y}_c(\mathrm{1X}) = \mathrm{X0}$. But the value of o is unconstrained by the second state, XX, in the sequence σ. On the other hand, a trajectory can still over-approximate real circuit execution. For example, the partial sequence $\sigma = \mathrm{XX}, \mathrm{X0}, \ldots$ can begin a trajectory, because $\mathrm{Y}_c(\mathrm{XX}) = \mathrm{XX} \sqsubseteq \mathrm{X0}$. But this sequence does not constrain node i to be 1 in the first state. Of course, any sequence completely free of Xs is a trajectory exactly when it represents an actual Boolean execution trace of the circuit being modelled.

25.4 Trajectory Evaluation Logic

One of the keys to the efficiency of STE and its success with data-path circuits is its restricted temporal logic, which we now define. Certain formulas in this logic contain, as sub-formulas, expressions of ordinary propositional logic with (free) Boolean variables. Let P range over propositional formulas whose free variables are drawn from some set $\mathscr{V}$. A *trajectory formula* is a simple linear-time temporal

logic formula with the following syntax:

$$
\begin{array}{lll}
f := n \text{ is } 0 & & \text{—node } n \text{ has value } 0 \\
\quad | \;\; n \text{ is } 1 & & \text{—node } n \text{ has value } 1 \\
\quad | \;\; f \text{ and } g & & \text{—conjunction of formulas} \\
\quad | \;\; \mathsf{N}\, f & & \text{—} f \text{ holds in the next time step} \\
\quad | \;\; P \rightarrowtriangle f & & \text{—} f \text{ is asserted only when } P \text{ is true}
\end{array}
$$

where f and g range over formulas and $n \in \mathcal{N}$ ranges over the nodes of the circuit. The formula P is called a *guard* and may contain propositional variables in $\mathcal{V}$. The role of these variables will be discussed later—for now, it is helpful to note that they are distinct from the names of circuit nodes in the set $\mathcal{N}$.

The basic trajectory formulas 'n is 0' and 'n is 1' say that the circuit node n has value 0 or value 1, respectively. The operator and forms the conjunction of trajectory formulas. The trajectory formula $\mathsf{N}\, f$ says that the trajectory formula f holds at the next point of time. It is easy to see that any formula containing only these first four syntactic constructs simply asserts the presence of 0 or 1 at certain fixed points of time on the specified circuit nodes. For example,

$$\mathsf{clk} \text{ is } 0 \text{ and } \mathsf{N}\big(\mathsf{clk} \text{ is } 1 \text{ and } \mathsf{N}\big(\mathsf{clk} \text{ is } 0 \text{ and } \mathsf{N}(\mathsf{clk} \text{ is } 1)\big)\big)$$

describes two cycles of a clock clk that starts off low. It is obvious that the next-time operator distributes over conjunction.

The final construct $P \rightarrowtriangle f$ weakens the sub-formula f by requiring it to be satisfied only when the guard P is true. In essence, a trajectory formula represents a whole *set* of assertions about the presence of the Boolean values 0 and 1 on particular circuit nodes over time. A guard is a propositional formula that may contain Boolean variables, and a trajectory formula $P \rightarrowtriangle f$ with a guard P asserts f only for satisfying assignments of values to the Boolean variables in P. So for any trajectory formula, each assignment of values to the variables in all its guards gives a (possibly different) assertion about the presence of 0s and 1s on certain circuit nodes at particular points in time.

A trivial example is this trajectory formula:

$$x \rightarrowtriangle (\mathsf{a} \text{ is } 0 \text{ and } \mathsf{b} \text{ is } 1) \;\text{ and }\; \overline{x} \rightarrowtriangle (\mathsf{a} \text{ is } 1 \text{ and } \mathsf{b} \text{ is } 0).$$

The guards here are just the literals x and $\overline{x}$. The formula encodes two of the four possible input bit patterns we might present to the unit-delay AND-gate in Fig. 1, namely the ones in which a and b are mutex. More generally, the guards in a trajectory formula can be any propositional logic expressions and can have variables in common. This gives STE the expressive power to represent inter-dependencies among node values—and, as will be seen, to encode families of state abstractions for efficient model checking.

We can associate an arbitrary propositional formula P with a circuit node using the construct 'n is P' defined by

$$n \text{ is } P \;\triangleq\; (P \rightarrowtriangle n \text{ is } 1) \text{ and } (\overline{P} \rightarrowtriangle n \text{ is } 0)$$

We suppose that $\rightarrowtriangle$ binds more tightly than and, so may drop the brackets:

$$n \text{ is } P \triangleq P \rightarrowtriangle n \text{ is } 1 \text{ and } \overline{P} \rightarrowtriangle n \text{ is } 0$$

In the simplest case, we use this construction to attach distinct Boolean variables to input nodes for direct symbolic simulation of circuits. For example, we might assert the formula 'a is x and b is y' for our unit-delay AND-gate. Here, the variables x and y just name the values present on the input nodes a and b. We might expect, after simulation, the circuit model to satisfy '$\mathsf{N}(\mathsf{c}$ is $x \wedge y)$'.

25.4.1 Correctness Properties for Verification

Circuit correctness in symbolic trajectory evaluation is stated by *trajectory assertions* of the form $A \Rightarrow C$, where A and C are trajectory formulas. The intuition is that the *antecedent* A provides stimuli to circuit nodes and the *consequent* C specifies the values expected on circuit nodes as a response, after simulation. We might, for example, write

$$\mathsf{a} \text{ is } x \text{ and } \mathsf{b} \text{ is } y \;\Rightarrow\; \mathsf{N}(\mathsf{c} \text{ is } x \wedge y)$$

for a direct, exhaustive simulation of our little AND-gate. Using guards, this single property encodes all the four possible bit-patterns that might be presented on the two inputs a and b. To check just the mutex cases mentioned earlier, we write

$$x \rightarrowtriangle (\mathsf{a} \text{ is } 0 \text{ and } \mathsf{b} \text{ is } 1) \text{ and } \overline{x} \rightarrowtriangle (\mathsf{a} \text{ is } 1 \text{ and } \mathsf{b} \text{ is } 0) \;\Rightarrow\; \mathsf{N}(\mathsf{c} \text{ is } 0)$$

Here there is not such a direct correspondence between 'Boolean variables' and 'values on circuit nodes'. The variable x doesn't correspond to an input value at all, but just enumerates the two input stimuli we wish to consider. In fact, because x doesn't appear in the consequent, this formula effectively expresses a disjunction in the antecedent. It is essential for a full understanding of STE to bear in mind that the Boolean variables used in guards, in the general case, need not correspond to node values. They are a case-enumeration mechanism, situated a layer above circuit values.

We now say what it means for a sequence σ to entail a trajectory formula f. Entailment is defined with respect to an assignment of Boolean truth-values to the Boolean variables in the guards of the formula. Suppose $\mathscr{V}$ is the set of all such variables and $\phi \in \mathscr{V} \rightarrow \{\mathrm{T}, \mathrm{F}\}$ is an assignment of values to them. Define

$$\begin{aligned}
\sigma \models_\phi n \text{ is } 0 &\triangleq \begin{cases} \sigma(0) = \top, \text{ or} \\ \sigma(0) \in \mathscr{N} \rightarrow \mathscr{D} \text{ and } \sigma\ 0\ n = 0 \end{cases} \\
\sigma \models_\phi n \text{ is } 1 &\triangleq \begin{cases} \sigma(0) = \top, \text{ or} \\ \sigma(0) \in \mathscr{N} \rightarrow \mathscr{D} \text{ and } \sigma\ 0\ n = 1 \end{cases} \\
\sigma \models_\phi f \text{ and } g &\triangleq \sigma \models_\phi f \text{ and } \sigma \models_\phi g \\
\sigma \models_\phi \mathsf{N}\, f &\triangleq \sigma^1 \models_\phi f \\
\sigma \models_\phi P \rightarrowtriangle f &\triangleq \phi \Vdash P \text{ implies } \sigma \models_\phi f
\end{aligned}$$

where $\phi \Vdash P$ means that the propositional formula P is satisfied by the assignment ϕ of truth-values to the Boolean variables in P. It follows that if $\sigma_1 \models_\phi f$ and $\sigma_1 \sqsubseteq \sigma_2$ then $\sigma_2 \models_\phi f$. That is, if a sequence σ_1 entails f, then any sequence σ_2 consistent with σ_1 but having 'more information' about node values also entails f. Informally, we can think of both trajectory formulas and sequences as predicates on sets of circuit states. And both $\models_\phi$ and $\sqsubseteq$ (backwards) as forms of entailment.

A trajectory assertion $A \Rightarrow C$ is true for a given ϕ exactly when every trajectory of the circuit that entails A also entails C. For a given circuit c, define $\models_\phi A \Rightarrow C$ to mean that for all $\sigma \in \mathscr{T}_c$, if $\sigma \models_\phi A$ then $\sigma \models_\phi C$. Intuitively, saying $\sigma \in \mathscr{T}_c$ means σ is an abstraction (over-approximation) of a set of actual execution traces of the circuit. For a given ϕ, the antecedent A describes a certain finite scattering of 0s and 1s across selected circuit nodes over a bounded period of time. Every trajectory $\sigma \in \mathscr{T}_c$ that conforms to this description must also exhibit the scattering of 0s and 1s expected by the consequent C.

The use of propositional formulas as guards enables a single trajectory assertion $A \Rightarrow C$ to encode a whole family of stimulus-response correctness properties, one for each ϕ. An example is the partial correctness specification for the unit-delay AND-gate (Fig. 1) discussed earlier:

$$x \rightarrow (\mathsf{a\ is\ 0\ and\ b\ is\ 1})\ \text{ and }\ \overline{x} \rightarrow (\mathsf{a\ is\ 1\ and\ b\ is\ 0})\ \Rightarrow\ \mathsf{N(c\ is\ 0)}.$$

When $\phi(x) = \mathrm{T}$, this property enforces correctness for the specific case where the input values are $\mathsf{a} = 0$ and $\mathsf{b} = 1$. When $\phi(x) = \mathrm{F}$, it enforces correctness for the case where $\mathsf{a} = 1$ and $\mathsf{b} = 0$. As will be seen, the model-checking algorithm for STE treats this dependence on ϕ symbolically, verifying all the cases simultaneously. We write $\models A \Rightarrow C$ to mean that $\models_\phi A \Rightarrow C$ holds for all ϕ.

It can be seen from the semantics just given that STE is fundamentally a logic of *forward* propagation of information about circuit values over time. We might hope the following trajectory assertion would hold of the little unit-delay AND-gate in Fig. 1:

$$\mathsf{N(c\ is\ 1)}\ \Rightarrow\ \mathsf{a\ is\ 1\ and\ b\ is\ 1}$$

The output c cannot possibly be high, unless both inputs a and b were high on the last clock cycle. Observe, however, that $\sigma = \mathrm{XXX}, \mathrm{XX1}, \mathrm{XXX}, \mathrm{XXX}, \ldots$ is a trajectory of this circuit that satisfies the antecedent but not the consequent. So this property is false. A sketch of what it would take to have the semantics of a 'bidirectional STE' in which this property could hold is given by Roorda in [53]. The fundamentally forwards nature of the STE logic flows from its basis as a logic of (forwards) circuit simulation. Roorda and Claessen have proposed to use SAT to encode the circuit along with the verification in an STE model-checking run, allowing backward information propagation.

We also note in passing that it is possible for an antecedent to be unsatisfiable, either outright or by all (or just some) real circuit executions. This is the problem of *antecedent failure* [7] or, more generally, *vacuity* in logic-based verification [8]. The most obvious case is where the antecedent places inconsistent values on some

circuit node, as for example in

$$x \rightarrowtriangle \mathsf{a\ is\ 0\ and}\ y \rightarrowtriangle \mathsf{a\ is\ 1}.$$

This is inconsistent when ϕ assigns the truth-value T to x and y. Another case is when the antecedent places a value on a circuit node that disagrees with what circuit execution itself produces. (For this to happen, the node in question must be the output of a state-holding element, not a primary input.) The semantic representation for the circuit state arising in the presence of such an inconsistency is the special top state, $\top$, introduced in Sect. 25.3.1.

Both types of antecedent failure can be detected by STE implementations and an error raised or warning given. Typically, a practitioner will then investigate the cause and rewrite the offending antecedent. Sometimes antecedent failure can safely, if cautiously, be ignored because it is known that cases in which inconsistency occurs do not to arise in practice. A more subtle problem is so-called *hidden vacuity* [69]. This can occur when the antecedent requires some specific value, 1 say, on a particular node that in any actual circuit execution would be 0, but which is assigned X by the circuit model because of incomplete information. For a full analysis and some methods to detect vacuity in STE, including the hidden type, see [69].

25.4.2 Correctness Properties Under Assumptions

It is convenient to have a notation that restricts the valuations ϕ for which a correctness property $\models_\phi A \Rightarrow C$ is verified to those that satisfy some given predicate. If P is a formula of propositional logic, we write $P \vdash A \Rightarrow C$ to mean that $\models_\phi A \Rightarrow C$ holds for all ϕ that satisfy P. This is used extensively in practice to express environmental constraints or assumptions on input values represented directly by Boolean variables. The verification script for an industrial-scale sub-circuit may well have hundreds of lines of text generating such formulas in order to accurately characterise the complex environment in which it operates.

Note that antecedents alone can also encode assumptions: writing the antecedent '$\mathsf{a\ is}\ x\ \mathsf{and\ b\ is}\ x$' makes the implicit assumption that two input nodes a and b have the same value. A methodology used by Intel for structuring both kinds of assumptions and translating them into System Verilog Assertions is presented in [44]. This supports a form of assume-guarantee reasoning; blocks of circuitry are verified in STE under assumptions about the operating environment, which are then translated into System Verilog Assertions and checked with simulation.

25.4.3 Internal Combinational Circuitry

It remains to note that what real implementations of STE do is not fully reflected in the simple story presented above. We have assumed throughout, and in particular in our formulation of next-state functions, that models and formulas refer only

to circuit nodes that are primary inputs or outputs of state-holding elements. But, as mentioned earlier, STE trajectory formulas can also mention internal nodes of combinational logic. A precise formulation of the semantics of this is a little involved [54], but the intuition is roughly the following. Suppose n is an internal node, driven by some fan-in cone of combinational logic. The sub-formula 'n is 1' should hold exactly when forward propagation through this logic of all known information about other node values in the circuit implies that n must have value 1. The practical value of this capability is that it exposes the fine detail of internal values within complex circuits to inspection and debugging by STE model checking.

25.5 The Fundamental Theorem of Trajectory Evaluation

A key property of trajectory evaluation logic is that for any trajectory formula f there exists a unique weakest sequence that entails f (assuming a fixed assignment ϕ of Boolean values to variables). This is called the *defining sequence* for f and is written $[f]^\phi$. It is defined by recursion over the syntax of trajectory formulas. Recall that a sequence is an element of $\mathbb{N} \to ((\mathscr{N} \to \mathscr{D}) \cup \{\top\})$; in the definition that follows, $t \in \mathbb{N}$ ranges over points in time and $n \in \mathscr{N}$ ranges over node names.

$$
\begin{array}{ll}
[m \text{ is } 0]^\phi & t\ n \triangleq 0 \text{ if } m{=}n \text{ and } t{=}0, \text{ otherwise X} \\
[m \text{ is } 1]^\phi & t\ n \triangleq 1 \text{ if } m{=}n \text{ and } t{=}0, \text{ otherwise X} \\
[f \text{ and } g]^\phi\ t & \triangleq ([f]^\phi\ t) \sqcup ([g]^\phi\ t) \\
[\mathsf{N}\ f]^\phi & t\ n \triangleq [f]^\phi\ (t{-}1)\ n \text{ if } t{\neq}0, \text{ otherwise X} \\
[P \rightarrowtriangle f]^\phi & t\ n \triangleq [f]^\phi\ t\ n \text{ if } \phi \Vdash P, \text{ otherwise X.}
\end{array}
$$

Note that, in the clause for and, we take the lattice join ($\sqcup$) of states.

The crucial property enjoyed by this definition is that $[f]^\phi$ is the unique weakest sequence that satisfies f for the given ϕ. That is, for fixed ϕ and any trajectory σ, we have that $\sigma \models_\phi f$ if and only if $[f]^\phi \sqsubseteq \sigma$. STE enjoys this property only because the language of trajectory formulas is monotonic—the defining sequence would not exist if the logic included negation or disjunction. (But see [32] for a more general presentation that does include negation, and a strong 'until' operator too.)

We can also define the weakest *trajectory* that satisfies each formula, for a given ϕ. Recall that a trajectory is a sequence of abstract states that respects the next-state function of the circuit—an abstract constraint on sequences of circuit states that is consistent with the circuit function. The *defining trajectory* for a formula, written $[\![f]\!]^\phi$, is given by the following recursive calculation over time points:

$$
\begin{array}{ll}
[\![f]\!]^\phi\ 0 & \triangleq [f]^\phi\ 0 \\
[\![f]\!]^\phi\ (t{+}1) & \triangleq [f]^\phi\ (t{+}1) \sqcup \mathrm{Y}_c([\![f]\!]^\phi\ t).
\end{array}
\tag{1}
$$

The defining trajectory of a formula f is its defining sequence with the added constraints on state transitions imposed by the circuit, as modelled by the next-state

function Y_c. The first state $[\![f]\!]^\phi\, 0$ is just given by any assumptions on node values at time 0 made by the antecedent. (Hence, in STE model checking, the initial state of a circuit is completely unknown by default. If we want to start in some specified initial state space, we have to characterise this explicitly in the antecedent.) Each successive state is obtained by combining, using the lattice join operator, any assumptions about node values at that time made by the antecedent with the constraints given by applying the model to the previous state. It is easy to see that folding in the effect of Y_c ensures that the sequence $[\![f]\!]^\phi$ is indeed a *trajectory* of the circuit c. Note also that if there is a conflict between the circuit and the antecedent at any point in time, then joining the two constraints will result in the top state $\top$.

It can be shown that $[\![f]\!]^\phi$ is the unique weakest trajectory that satisfies f, for the given ϕ. That is, for fixed ϕ and any trajectory $\sigma \in \mathscr{T}_c$, we have that $\sigma \models f$ if and only if $[\![f]\!]^\phi \sqsubseteq \sigma$.

The *Fundamental Theorem of Trajectory Evaluation* [64] follows immediately from the previously stated properties of $[f]^\phi$ and $[\![f]\!]^\phi$. It states that for a given ϕ, the trajectory assertion $\models_\phi A \Rightarrow C$ holds exactly when $[C]^\phi \sqsubseteq [\![A]\!]^\phi$. The intuition is that the weakest trajectory satisfying the antecedent must entail the defining sequence characterising the consequent: what the circuit 'does', under the assumptions made, must be among the things it is intended to do.

25.6 STE Model Checking

The Fundamental Theorem of Trajectory Evaluation gives a model-checking algorithm for trajectory assertions: to see whether $\models_\phi A \Rightarrow C$ holds for a given ϕ, just compute the sequences of states $[C]^\phi$ and $[\![A]\!]^\phi$ and compare them point-wise for every circuit node at every relevant point in time. This works because both A and C will have only a finite number of nested next-time operators, and so only finite initial segments of the defining trajectory and defining sequence need to be calculated and compared.

In practice, the defining trajectory of A is computed iteratively, and each resulting state is checked against the consequent as it is generated. In essence, the algorithm does a step-by-step unrolling of the recursive definition of $[\![A]\!]^\phi$ shown above as Eq. (1). Throughout the computation, the current state is maintained as an assignment of a value from $\mathscr{D}$ to each circuit node. Initially, all nodes are set to X. Any nodes assumed to have specific values by the antecedent A at time 0 are then updated to have these values. Any constraints on nodes at time 0 in the consequent can then be checked against these initial node values. At each subsequent step, the algorithm first computes the next state of the circuit from the current one, according to the next-state function Y_c. This is done in practice by three-valued simulation of a circuit net-list description [14], in which logic gates or other primitive circuit elements are interpreted to operate over the domain $\{X, 0, 1\}$. It is here, at the heart of STE, that propagation of 'information' about states is enacted. For example, the simulator's truth table for an AND-gate is shown in Fig. 3. Notice that the output is

$\wedge$	X	0	1
X	X	0	X
0	0	0	0
1	X	0	1

Fig. 3 Three-valued truth-table for an AND-gate

known to be 0 when any input is 0, even if another input is X. With some ingenuity, three-valued interpretations for simulation can be given to circuits in a range of design styles and at various abstraction levels—including, notably, transistors at the 'switch' level [15, 64]. Once the next state has been obtained, it is combined with any assumptions about circuit values in the current step stated by the antecedent. This is done by taking the lattice join ($\sqcup$) of the values, node by node—effectively calculating the join of the next state obtained from Y_c and the corresponding state in the defining sequence of the antecedent. (If there is a clash of values on any node, then the result is $\top$ and we have an antecedent failure.) Finally, any constraints on particular nodes at this time step in the consequent are checked against the calculated node values. The process continues for as many time steps as the deepest nesting of next-time operators in the trajectory assertion. If at any point there is a conflict between the consequent and a node value in the states being calculated, the property has been falsified. If they always agree, the property has been proved.

As already noted, all information about node values in the 'initial state' of an STE verification comes from only the antecedent of the property checked: each node starts out X, unless the antecedent explicitly assigns 0 or 1. A subtle issue of large practical significance is that the verification engineer must choose which input and state nodes to 'drive' via the antecedent, in order to produce a determinate circuit response to the antecedent stimulus. If too few input nodes are driven, Xs propagate into the simulation and the result fails to satisfy the consequent. The appearance of X on a node where the consequent expects a 0 or 1 is known as a *weak disagreement*. In this case the property is neither falsified nor proved.

For complex or industrial-scale circuits, it can be quite hard to find a minimal set of input nodes that need to be stimulated by the antecedent to get a determinate result—to eliminate weak disagreements. Doing this is important because it increases the abstraction level and hence tractability of the verification, especially in the symbolic algorithm discussed in Sect. 25.6.1. The Forte methodology has a whole phase of work for this activity, dubbed *wiggling* by Robert Jones, and the Forte environment has specialised computational tools to support it [1, 38]. Some research on automated methods for refining STE properties to eliminate unwanted Xs is reported in [18, 55, 69].

As briefly mentioned earlier, properties in full STE can refer to internal nodes of combinational logic, as well as primary inputs and the outputs of state-holding elements. In the simulation engine that underlies STE there is, therefore, a phase during which values asserted on internal nodes by antecedents are propagated through this logic, before the consequent is checked. This can involve computing fixed points, for certain models of some circuit design styles.

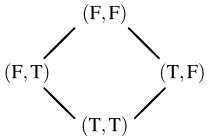


Fig. 4 Lattice of node values, encoded by pairs of Booleans

25.6.1 The Symbolic Model-Checking Algorithm

The model-checking algorithm just sketched requires ϕ to be supplied: given a fixed assignment ϕ of values to variables in the guards, we calculate and compare $[C]^\phi$ and $[\![A]\!]^\phi$ point-wise. But much of the power of STE comes from the key observation that it is not necessary to supply ϕ in advance; instead, the comparison can be computed for all possible valuations, i.e. parametrically in ϕ. The result is a constraint on ϕ—a formula called a *residual*—that gives precisely the condition on Boolean variables in the guards under which the trajectory assertion $A \Rightarrow C$ is verified. If the residual is T, the circuit satisfies all the properties encoded by this assertion. If it is F, the circuit satisfies none of them. Otherwise, the residual encodes the class of all variable assignments under which an STE trajectory assertion $A \Rightarrow C$ holds. In other words, a residual is the weakest constraint R such that $R \models A \Rightarrow C$, in the notation of Sect. 25.4.2. This provides an enormously valuable and flexible debugging aid because, with R in hand, a verification engineer can explore the whole counterexample space computationally. The set of all counterexamples is encoded by $\overline{R}$.

The symbolic STE model-checking algorithm works as follows. At the level of basic data values in $\{\mathrm{X}, 0, 1\}$, the required computation is to show that

$$[C]^\phi\ t\ n \leq [\![A]\!]^\phi\ t\ n \tag{2}$$

for all $t \geq 0$ and $n \in \mathscr{N}$. For each circuit node at each relevant point in time, we compare the value expected by the consequent to that given by the antecedent and circuit simulation. To make this comparison 'parametric' in ϕ, we use a pair of Boolean formulas to encode functions from ϕ to data values in $\mathscr{D}$ and do the comparison on this encoding.

The encoding is the following. First, enhance the three-valued set $\mathscr{D} = \{\mathrm{X}, 0, 1\}$ with an additional 'top' element, Z, that represents the value of an over-constrained node. This is not really a fundamental extension; any sensible implementation will raise an exception if any node individually becomes Z, so any state with one or more nodes being Z in effect represents the top state $\top$. Now encode the four values $\{\mathrm{X}, 0, 1, \mathrm{Z}\}$ by pairs of Booleans in $\mathbb{B} \times \mathbb{B}$, as follows:

$$\mathrm{X} \triangleq (\mathrm{T}, \mathrm{T}) \qquad 0 \triangleq (\mathrm{F}, \mathrm{T}) \qquad 1 \triangleq (\mathrm{T}, \mathrm{F}) \qquad \mathrm{Z} \triangleq (\mathrm{F}, \mathrm{F})$$

Then define an information ordering $\leq$ on these values to make it a lattice as in Fig. 4. Formally, define $(a_1, a_2) \leq (b_1, b_2)$ to hold exactly when b_1 implies a_1

∧	X	0	1	Z
X	X	0	X	0
0	0	0	0	0
1	X	0	1	Z
Z	0	0	Z	Z

Fig. 5 Four-valued truth-table for conjunction

and b_2 implies a_2. It is easy to check that this definition captures the Hasse diagram in Fig. 4. The join of two elements is equally straightforward and is given by component-wise conjunction: $(a_1, a_2) \sqcup (b_1, b_2)$ equals (a_1 and b_1, a_2 and b_2).

This is the so-called 'dual-rail' encoding of four-valued logic employed in STE implementations [62]. It is straightforward to define logical operators over these values in terms of ordinary logical operations on the component Booleans. For example, the four-valued truth-table for conjunction is given by Fig. 5 and can be defined by

$$(a_1, a_2) \wedge (b_1, b_2) \stackrel{\triangle}{=} (a_1 \text{ and } b_1,\ a_2 \text{ or } b_2).$$

As will be seen below, the underlying simulation engine of full symbolic STE interprets circuit net-lists in a symbolic version of this four-valued logic.

Now, a valuation is a function in $\mathcal{V} \to \mathbb{B}$, from variables $\mathcal{V}$ to the Boolean truth-values $\mathbb{B}$. We want the node values computed by our model-checking algorithm to depend on a valuation, so the value associated with each node will now be a function from valuations to (encoded) lattice elements:

$$(\mathcal{V} \to \mathbb{B}) \to (\mathbb{B} \times \mathbb{B}).$$

Any such function f can be represented by a pair of formulas of propositional logic (P_1, P_2). Define P_1 so that it is satisfied by just those valuations that f maps to (T, a_2) for some a_2. Likewise, define P_2 so that it is satisfied by just those valuations that f maps to (a_1, T) for some a_1. In a nutshell, $(\mathcal{V} \to \mathbb{B}) \to (\mathbb{B} \times \mathbb{B})$ is isomorphic to $((\mathcal{V} \to \mathbb{B}) \to \mathbb{B}) \times ((\mathcal{V} \to \mathbb{B}) \to \mathbb{B})$, and so a function from valuations to Booleans is straightforwardly represented by a pair of propositional formulas with free variables.

Given two functions f and g in this representation, we can compute a condition that characterises the valuations ϕ for which $f\ \phi \leq g\ \phi$. Suppose f is represented by the pair of formulas (P_1, P_2) and g is represented by the pair of formulas (Q_1, Q_2). Simply form the following formula of propositional logic:

$$(Q_1 \supset P_1) \wedge (Q_2 \supset P_2), \tag{3}$$

corresponding to the definition of the ordering $\leq$ just given. The result is a propositional formula satisfied precisely by the valuations ϕ for which $f\ \phi \leq g\ \phi$.

Using this idea, the symbolic version of STE proceeds just as indicated in Sect. 25.6, except that node values are now computed parametrically in the valuation. The algorithm maintains a pair of formulas for each circuit node, representing the value on each node as a function of an arbitrary valuation. Initially, every circuit node gets the dual-rail value (T, T), representing X for every valuation. Node val-

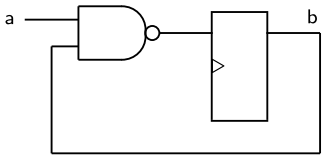


Fig. 6 A simple state-holding circuit

ues are then updated, step by step, to new symbolic dual-rail values according to the general scheme already explained. Note that values coming from the antecedent and consequent are now conditional on guards that constrain variables and so also represented by the dual-rail encoding. For example, if the antecedent says '$P \rightarrow \mathsf{n}$ is 1', this becomes the dual-rail value $(\mathrm{T}, \overline{P})$. Under any valuation that satisfies P, this equals (T, F), the encoding for 1; under any other valuation, it equals (T, T), the encoding for X. The circuit net-list is also interpreted over this dual-rail parametric representation of four-valued logic. For example, if (P_1, P_2) and (Q_1, Q_2) are the dual-rail values currently on the two inputs of an AND gate, then the output is given by the pair of formulas $(P_1 \wedge Q_1, P_2 \vee Q_2)$. Lattice joins are also computed symbolically—by component-wise conjunction, as before. Finally, whenever a comparison is made between a computed node value and a value specified by the consequent, a formula of propositional logic is built of the form shown in Eq. (3). The conjunction of all such formulas is the residual delivered as the final outcome of model checking.

We note in passing that weak disagreements in this symbolic version of STE can be conditional on the valuation parametrically encoded in dual-rail values. That is, there may be a weak disagreement between the computed value of a node and the consequent's specification of its required value for only *some* valuations of the Boolean variables in play. This adds an extra dimension of complication and subtlety to the process of eliminating unwanted Xs and tools that support it.

In implementations of STE, the propositional formulas of the dual-rail encoding are represented by BDDs, or by some non-canonical representation such as AIGs (and-inverter graphs [33]). In the latter case, SAT technology can be used to check the final results of STE verification runs. Intel's Forte system has full support for both approaches.

25.6.2 Some Small Examples in Detail

We now illustrate the symbolic algorithm, in detail, by showing its calculations for a few properties of the little example circuit in Fig. 6.

Suppose we want to use STE to perform a full verification of this device by symbolic simulation. We introduce two Boolean variables, x and y, to stand for the initial values of the input node a and state node b, respectively. The trajectory assertion we want to check is

$$\mathsf{a}\ \mathsf{is}\ x\ \mathsf{and}\ \mathsf{b}\ \mathsf{is}\ y \Rightarrow \mathsf{N}(\mathsf{b}\ \mathsf{is}\ \overline{x} \vee \overline{y}).$$

This employs a distinct, unconstrained Boolean variable to name the value on each node initially, and then checks that the expected function of these values appears on node b at the next cycle.

Model checking begins by assigning the dual-rail value $\mathrm{X} = (\mathrm{T}, \mathrm{T})$ to all nodes, and then overlaying the assumptions made by the antecedent at time 0 onto this initial state. Recall that 'n is P' abbreviates '$P \rightarrow \mathsf{n}$ is 1 and $\overline{P} \rightarrow \mathsf{n}$ is 0'. So the antecedent really makes two guarded assertions about node a, encoded by dual-rail values as follows:

$$x \rightarrow \mathsf{a} \text{ is } 1, \text{ encoded by } (\mathrm{T}, \overline{x}) \quad \text{and} \quad \overline{x} \rightarrow \mathsf{a} \text{ is } 0, \text{ encoded by } (x, \mathrm{T}).$$

The semantics of trajectory formulas for and takes the lattice join of states, so the overall assumption made by the antecedent about node a is

$$(\mathrm{T}, \overline{x}) \sqcup (x, \mathrm{T}) = (\mathrm{T} \wedge x, \overline{x} \wedge \mathrm{T}) = (x, \overline{x}).$$

Likewise, the dual-rail encoding of the antecedent's assumptions about node b are given by $(y, \overline{y})$. State 0 of the defining trajectory for the antecedent, τ say, is therefore given by

$$\tau\ 0\ \mathsf{a} = (x, \overline{x}) \quad \text{and} \quad \tau\ 0\ \mathsf{b} = (y, \overline{y})$$

Notice that neither of these nodes is ever X—for every valuation, we get either $(\mathrm{T}, \mathrm{F}) = 1$ or $(\mathrm{F}, \mathrm{T}) = 0$, so there is going to be no abstraction in this verification. In general, if the two 'rails' of a dual-rail value are the negation of each other, the encoded function on valuations never yields lattice values X or Z.

Now circuit simulation computes the state at time 1. The conjunction of values on a and b, according to the truth table given earlier, is encoded by $(x \wedge y, \overline{x} \vee \overline{y})$. Negation of dual-rail values is done simply by swapping the components of the pair, so the state of the circuit at time 1 is given by

$$\tau\ 1\ \mathsf{a} = (\mathrm{T}, \mathrm{T}) \quad \text{and} \quad \tau\ 1\ \mathsf{b} = (\overline{x} \vee \overline{y}, x \wedge y)$$

Notice that a has been reset to X, because it is a primary input and therefore free to take on any new value at each cycle. The antecedent makes no assumptions about values at time 1, so this is also the state at time 1 in the defining trajectory.

It remains to check this state against the expectations of the consequent—i.e. state 1 of the defining sequence of the consequent. It is easy to see that the dual-rail value that encodes 'b is $\overline{x} \vee \overline{y}$' exactly matches the computed value just calculated for b in state 1 of the defining trajectory. Plugging both values into Eq. (3) yields a tautology, so the residual is equivalent to T and the property is satisfied outright.

In this example, there is no abstraction and the property holds for all valuations. It is instructive to sketch—very briefly—examples that illustrate abstraction and a non-constant residual. For the first, consider the property

$$x \rightarrow \mathsf{a} \text{ is } 0 \text{ and } \overline{x} \rightarrow \mathsf{b} \text{ is } 0 \Rightarrow \mathsf{N}(\mathsf{b} \text{ is } 1).$$

This verifies all the input cases of our little circuit for which the next state of b is 1. This time, the propositional variable x does not straightforwardly correspond to 'the value on an input', but serves as a symbolic 'index' to enumerate the two input stimuli we want to cover.

It is easy to see the first state of the defining trajectory, τ, for this example is

$$\tau\ 0\ \mathsf{a} = (\overline{x}, \mathrm{T}) \quad \text{and} \quad \tau\ 0\ \mathsf{b} = (x, \mathrm{T})$$

There is abstraction here. Node b is X for valuations that set x to true, and node a is X for valuations that set x to false. After four-valued symbolic simulation, state 1 of the defining trajectory is found to be

$$\tau\ 1\ \mathsf{a} = (\mathrm{T}, \mathrm{T}) \quad \text{and} \quad \tau\ 1\ \mathsf{b} = (\mathrm{T} \vee \mathrm{T}, \overline{x} \wedge x)$$

The dual-rail encoding of values on node b simplifies to (T, F), which encodes the constant lattice value 1. This agrees with the consequent, and so the property holds.

To see a non-constant residual, consider the following property:

$$\mathsf{a\ is}\ x\ \mathsf{and\ b\ is}\ y \Rightarrow \mathsf{N(b\ is\ 1)}.$$

We have again 'driven' both nodes a and b with symbolic Boolean values x and y, respectively. As before, state 1 of the defining trajectory will be

$$\tau\ 1\ \mathsf{a} = (\mathrm{T}, \mathrm{T}) \quad \text{and} \quad \tau\ 1\ \mathsf{b} = (\overline{x} \vee \overline{y}, x \wedge y)$$

To get the residual, we compare this, node-by-node, to state 1 of the defining sequence, σ, of the consequent:

$$\sigma\ 1\ \mathsf{a} = (\mathrm{T}, \mathrm{T}) \quad \text{and} \quad \sigma\ 1\ \mathsf{b} = (\mathrm{T}, \mathrm{F})$$

We conjoin the results. Calculating with the symbolic version of $\leq$ gives:

$$\text{Node a}: \ (\mathrm{T}, \mathrm{T}) \leq (\mathrm{T}, \mathrm{T}) = (\mathrm{T} \supset \mathrm{T}) \wedge (\mathrm{T} \supset \mathrm{T}) = \mathrm{T}$$

$$\text{Node b}: \ (\mathrm{T}, \mathrm{F}) \leq (\overline{x} \vee \overline{y}, x \wedge y) = (\overline{x} \vee \overline{y} \supset \mathrm{T}) \wedge (x \wedge y \supset \mathrm{F}) = \overline{x} \vee \overline{y}$$

The overall residual is therefore '$\overline{x} \vee \overline{y}$', representing the precise set of cases for which the circuit satisfies the specification. It is easy to see that this is right: node b will be 1 after one clock cycle exactly when at least one of a and b starts off 0.

25.6.3 Model Checking Properties Under Assumptions

Few industrial verifications take place in isolation from environmental and other operating assumptions about the blocks of circuitry under inspection. In practice, the properties to be verified in STE are virtually always of the form $P \models A \Rightarrow C$,

where P expresses some potentially complex assumptions about the environment the circuit will operate in. Typically, there are a number of primary inputs directly driven by distinct Boolean variables in the antecedent, and the formula P says that these values satisfy some relational assumptions [44]. For example, when verifying a floating-point arithmetic unit, it may (and, for correctness, probably must) be assumed that the vectors of Boolean variables representing the incoming operands conform to the format laid down by the IEEE 754 floating-point standard. When verifying normal modes of operation, it will have to be assumed that any reset or test-scan inputs are tied to the inactive state, and so on. Constraints may also arise from case-splitting strategies that partition the input space into tractable sub-spaces [65].

Conditional verifications like these are efficiently handled in STE as follows. Suppose the assumption $P[x]$ constrains some Boolean variables $\vec{x} = x_1, \ldots, x_n$ that occur in a trajectory assertion $A[\vec{x}] \Rightarrow C[\vec{x}]$. One obvious but inefficient way to establish this assertion is to use STE to obtain a residual and then check that $P[\vec{x}]$ implies this. But this is usually not practical. If BDDs are used, it may be too complex to compute the defining trajectory for $A[\vec{x}]$ with a symbolic simulator, for unrestricted $\vec{x}$. We might be able to get around this by using a non-canonical form such as AIGs for the formulas in our dual-rail values. But then it may anyway be intractable to check that $P[x]$ implies the residual, either by SAT or any other means.

A better way is to evaluate the defining trajectory only for variable assignments that actually do satisfy $P[\vec{x}]$. This can be done by using a *parametric representation* of $P[\vec{x}]$ to incorporate the assumptions this formula makes into the model-checking calculations of STE. Given a satisfiable $P[\vec{x}]$, we compute a vector of n propositional formulas $\vec{Q} = \text{param}(P[\vec{x}], \vec{x})$ that are substituted for the variables $\vec{x}$ in the original trajectory assertion. These formulas contain fresh Boolean variables, distinct from those in $\vec{x}$, and are constructed so that $P[\vec{Q}/\vec{x}]$ is a tautology. Moreover, for any assignment of truth-values to $\vec{x}$ that satisfies $P[\vec{x}]$, there is some valuation of all the variables in $\vec{Q}$ under which assigning the *truth-value* of each formula Q_i to the corresponding variable x_i, for $1 \leq i \leq n$, gives this satisfying assignment to $\vec{x}$. An algorithm for computing the parametric representation of a formula and its correctness proof are given in [37].

These properties ensure that the original property $P[\vec{x}] \models A[\vec{x}] \Rightarrow C[\vec{x}]$ holds just when $\models A[\vec{Q}/\vec{x}] \Rightarrow C[\vec{Q}/\vec{x}]$ holds. They are equivalent, but the latter property has the assumption implicitly encoded into the guards of the antecedent and consequent and can be much easier to model check. A minor complication is that any non-constant residual resulting from $A[\vec{Q}/\vec{x}] \Rightarrow C[\vec{Q}/\vec{x}]$ will be a formula over the fresh variables in $\vec{Q}$, which are an encoding artifact unlikely to be meaningful to the verification engineer. But methods exist to map such residuals back into the original variable space for counterexample analysis.

In practice, parametric representation of assumption formulas is virtually essential to the use of STE on serious examples, and is extensively used to restrict verifications to a care set and to tackle complexity by input space decomposition [3, 36, 37]. The technique is independent of the symbolic simulation algorithm in STE, does not require modifications to the circuit, and can be used to constrain both input and state values, as well as the values of internal combinational logic.

25.7 Abstraction and Symbolic Indexing

The lattice of abstract sets of states underlying STE provides an integral and flexible way to control model-checking complexity. As with all abstraction mechanisms, the strategy is to provide a means to minimize the information about circuit operation that is computationally represented when verifying a property. Crudely speaking, complexity is controlled in STE by having as many Xs as possible in place of irrelevant information about selected node values, as circuit simulation proceeds.

This mechanism of complexity control is intimately connected with the encoding, using Boolean variables, of the families of abstractions of circuit states that are computed during simulation. As already explained, propositional guards occurring in trajectory formulas introduce a layer of symbolic representation above the level of abstractions, by means of which families of abstractions are checked parametrically. The abstractions are 'indexed' by the Boolean variables present, with each valuation of the variables giving a separate abstraction case. In this setting, controlling complexity through abstraction means, in essence, controlling the complexity of the parametric representation of node values computed during simulation. When BDDs are used for the formulas in the dual-rail encoding, this reduces the simulation-time complexity of the BDDs computed to verify a circuit property. When a non-canonical representation such as AIGs is used, this can reduce the complexity of checking the eventual propositional logic problem that is produced by an STE model-checking run.

The mechanism of indexing by Boolean variables is pervasive in STE. Even the commonplace verification idiom of representing 'the value on a wire' symbolically by a variable is achieved through it. The most straightforward way of using STE is direct symbolic simulation with a kind of 'semantic cone-of-influence reduction' given by X-propagation. The antecedent attaches a distinct Boolean variable to each primary input that 'matters' to the property and leaves the remaining nodes X. These input values, fully determined as Boolean and named symbolically by variables, are then propagated through the circuitry to produce symbolic circuit states that are tested against the consequent. Internal node values computed during simulation may be X for some valuations of the variables, provided all the nodes inspected by the consequent have the stipulated Boolean functions of the inputs.

This simplistic approach often does not scale to complex or large circuits, and further abstraction may have to be introduced to control the complexity of model checking. This is known as *weakening*. In essence, weakening consists in making interventions in the model-checking algorithm that result in the trajectory computed during symbolic simulation being weaker than the defining trajectory of the property being checked. If this weaker trajectory still entails the consequent, then the property still holds. If the verification fails with the weakened trajectory, then we can draw no conclusions about the original property.

More formally, recall that the Fundamental Theorem of Trajectory Evaluation says that, for a given ϕ, the assertion $\models_\phi A \Rightarrow C$ holds just when $[C]^\phi \sqsubseteq [\![A]\!]^\phi$. Now suppose we have some sequence $\sigma \sqsubseteq [\![A]\!]^\phi$ that is weaker than $[\![A]\!]^\phi$. This means that some nodes at some points in time are given values by σ that are lower

in the value lattice than those given by $[\![A]\!]^\phi$. Informally, the defining trajectory has been *weakened* by throwing away knowledge about some node values. If σ still entails the defining sequence of the consequent, that is if $[C]^\phi \sqsubseteq \sigma$ still holds, then we know that $[C]^\phi \sqsubseteq [\![A]\!]^\phi$ also holds and so $\models_\phi A \Rightarrow C$ is verified.

The justification just sketched extends to the parametrically encoded families of trajectories computed by the symbolic model-checking algorithm. Weakening in this case means setting the values of one or more nodes, at selected points in the simulation, to X for some or all valuations of the variables occurring in their dual-rail encoding. The dual-rail value (P_1, P_2) on a selected node at a certain point in time can simply be set to $\mathrm{X} = (\mathrm{T}, \mathrm{T})$ or, more generally, be moved arbitrarily 'closer' to X by setting it to any pair of formulas (Q_1, Q_2) for which $P_1 \supset Q_1$ and $P_2 \supset Q_2$. This makes the node have value X for more valuations of the Boolean variables at that point in time.

Forte provides fine-grained access to weakening by user-level directives that specify the nodes to weaken and—separately for each node—both the simulation times at which to weaken it and a condition on Boolean variables that stipulates for which valuations to weaken it. Users can therefore manually weaken individual nodes at arbitrary points of time during simulation and to arbitrary degrees, with a view to reducing the BDD complexity of their dual-rail values whilst retaining just enough information about node values to satisfy the consequent. This is safe, because the theory just sketched tells us that however a node's value is weakened during a verification, if the consequent is satisfied then the trajectory assertion being checked still holds. Weakening may, however, introduce hidden vacuity (discussed in Sect. 25.4.1).

A more automated mechanism is *dynamic weakening* [65], in which a complexity threshold is set to limit the size of the BDDs representing dual-rail values. If, at any point during simulation, the size of the dual-rail value for a particular node exceeds the threshold, it is simply set to the unknown value $\mathrm{X} = (\mathrm{T}, \mathrm{T})$. This works well when the BDDs of the internal nodes along the 'relevant paths' within some complex circuitry are well behaved, while the BDDs along other paths blow up in size. Some detailed and illuminating examples of how this mechanism can solve practical verification problems can be found in [65, pp. 1389–1390].

Intel's Core i7 processor verification effort used some more sophisticated automated weakening methods that compute 'causal fan-in information from a circuit trace to determine weakening points' [41]. The Intel engineers who conducted this impressive large-scale STE verification report that dynamic weakening together with these automated weakening techniques 'solve most circuit simulation capacity problems without need for human intervention'.

25.7.1 Symbolic Indexing

Symbolic indexing is the systematic use of weakening, controlled through the way in which antecedents are formulated, to perform partitioned abstraction that is highly

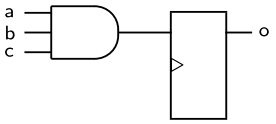


Fig. 7 Three-input, unit-delay AND-gate

effective for verifying certain regular or symmetric circuit structures. Like weakening, it is an implementation optimisation that reduces model-checking complexity. But instead of only controlling BDD sizes by driving node values towards X, it also exploits symmetry to reduce the number of Boolean variables needed to verify certain circuit properties.

The idea can be illustrated by the following trivial example. Consider a unit-delay AND-gate with three inputs as in Fig. 7. For the purpose of writing down states, we will order the nodes $\mathsf{a} < \mathsf{b} < \mathsf{c} < \mathsf{o}$. A direct STE verification that does not exploit the abstraction lattice is achieved by checking the following trajectory assertion:

$$\mathsf{a\ is}\ v_1\ \mathsf{and\ b\ is}\ v_2\ \mathsf{and\ c\ is}\ v_3 \Rightarrow \mathsf{N(o\ is}\ v_1 \wedge v_2 \wedge v_3). \tag{4}$$

This is just direct Boolean symbolic simulation. The antecedent attaches a distinct, unconstrained Boolean variable to each input node, and the consequent asserts that the expected function of these variables appears on the output.

We can be more clever than this by using STE's abstraction lattice to reduce the number of Boolean variables needed to verify this gate. The key observation is that if any one input is 0, then the output will be 0 regardless of the other inputs. We can exploit this to introduce X-abstraction in the model-checking run. There are four cases to check—three in which one of the inputs is known to be 0 and the others are unknown, and one in which all three inputs are known to be 1. We can enumerate or 'index' these with two Boolean variables, say x_1 and x_2. We write the following property:

$$\begin{array}{l} \overline{x_1} \wedge \overline{x_2} \rightarrow \mathsf{a\ is\ 0\ and} \\ x_1 \wedge \overline{x_2} \rightarrow \mathsf{b\ is\ 0\ and} \\ \overline{x_1} \wedge x_2 \rightarrow \mathsf{c\ is\ 0\ and} \\ x_1 \wedge x_2 \rightarrow \mathsf{a\ is\ 1\ and\ b\ is\ 1\ and\ c\ is\ 1} \\ \quad \Rightarrow \\ \mathsf{N}(\overline{x_1} \vee \overline{x_2} \rightarrow \mathsf{o\ is\ 0\ and}\ x_1 \wedge x_2 \rightarrow \mathsf{o\ is\ 1}). \end{array} \tag{5}$$

Model checking this with STE will simultaneously check all four cases, each with a different abstraction of the sets of states arising during circuit simulation. Since any property verified in STE with a node set to X also holds when the node is either 0 or 1, this verification covers all input cases and is complete.

Notice that some of the abstractions in this trivial example overlap. For example, when $x_1 = \mathrm{F}$ and $x_2 = \mathrm{F}$, the initial abstract state of the defining trajectory is 0XXX. The maximal set of Boolean states of which this is an abstraction is

$$\{0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111\}.$$

When $x_1 = \mathrm{T}$ and $x_2 = \mathrm{F}$, the initial abstract state of the defining trajectory is X0XX. The maximal set of Boolean states of which this is an abstraction is

$$\{0000, 0001, 0010, 0011, 1000, 1001, 1010, 1011\}.$$

The two sets have the nonempty intersection $\{0000, 0001, 0010, 0011\}$, comprising the Boolean initial circuit states covered by both abstractions. This overlapping of 'partitioned' abstraction cases is a characteristic of STE not shared by, for example, existential abstraction in CTL model checking.

Symbolic indexing finds its greatest utility in verification of regular memory structures, where it can significantly reduce the number of BDD variables required to encode data values [13, 52, 70]. Consider an $n \times m$-bit memory, with n words of memory and m bits per word. Suppose we want to verify the simple property that the read operation correctly returns the data word stored at any given address. We could write an STE property whose antecedent associates a Boolean variable $d_{i,j}$ with each bit of the initial state of the memory, for $1 \leq i \leq n$ and $1 \leq j \leq m$, and then presents a symbolic address given by a vector of Boolean variables $\vec{a}$ of length $l = \log_2 n$ on the read address input. The consequent would stipulate that the data delivered at the output is the word stored at the addressed location.

A little notation must be introduced before we can write this trajectory assertion formally. We first introduce the circuit nodes that are involved. Suppose the circuit nodes holding the individual bits of memory are systematically named '$\mathsf{mem}[i][j]$', where $1 \leq i \leq n$ identifies the word of memory and $1 \leq j \leq m$ is the bit position within the word. Let $\mathsf{mem}[i]$ be the vector of nodes $\mathsf{mem}[i][1], \ldots, \mathsf{mem}[i][m]$ for $1 \leq i \leq n$. Suppose also that the circuit nodes of the address input bits are given by the vector $\mathsf{addr} = \mathsf{addr}_1, \ldots, \mathsf{addr}_l$. Finally, suppose $\mathsf{out} = \mathsf{out}_1, \ldots, \mathsf{out}_m$ are the circuit nodes of the memory's m-bit data output.

We will need a way to associate a vector of Boolean variables, or more generally a vector of propositional expressions, with a vector of nodes. For any vector of circuit nodes $\mathsf{n} = \mathsf{n}_1, \ldots, \mathsf{n}_k$ and vector of propositional expressions $\vec{P} = P_1, \ldots, P_k$, both of any length $k \geq 1$, define

$$\mathsf{n} \text{ is } \vec{P} \;\triangleq\; \mathsf{n}_1 \text{ is } P_i \;\text{ and }\; \cdots \;\text{ and }\; \mathsf{n}_k \text{ is } P_k.$$

So 'n is $\vec{P}$' just asserts that each circuit node in the vector n has the value given by the respective Boolean expression in the vector $\vec{P}$.

As mentioned, the antecedent of our trajectory assertion will associate Boolean variables $d_{i,j}$ with the individual bits of memory state, where $1 \leq i \leq n$ is the memory location and $1 \leq j \leq m$ is the bit position within that location. To express this, we will write $\vec{d}_i = d_{i,1}, \ldots, d_{i,m}$ for the vector of m Boolean variables associated with the bits of the word stored at memory location i.

To formulate the consequent, we first need a way to state, in propositional logic, that the address input $\vec{a}$ has a specific value in the interval $[1, n]$. It is routine to encode this by propositional assertions '$\vec{a} \sim i$', where $1 \leq i \leq n$, that hold exactly when the address bits $\vec{a}$ form the unsigned binary number corresponding to the

integer $i-1$. We can then define propositional expressions $d_{\vec{a},j}$, for $1 \leq j \leq m$ that give the jth bit of the data word at the memory location addressed by $\vec{a}$. Define

$$d_{\vec{a},j} \stackrel{\triangle}{=} (\vec{a} \sim 1 \wedge d_{1,j}) \vee \cdots \vee (\vec{a} \sim n \wedge d_{n,j}).$$

We are now in a position to express the required trajectory assertion formally:

$$\begin{array}{l} \mathsf{addr\ is}\ \vec{a}\ \ \mathsf{and} \\ \mathsf{mem}_1\ \mathsf{is}\ \vec{d}_1\ \mathsf{and}\ \cdots\ \mathsf{and}\ \mathsf{mem}_n\ \mathsf{is}\ \vec{d}_n \end{array} \quad \Rightarrow \quad \mathsf{out\ is}\ d_{\vec{a},1}, \ldots, d_{\vec{a},m}.$$

The antecedent simply associates a Boolean variable with each bit of memory state and introduces a vector of Boolean variables to name the bits of the read address. The consequent says that each bit of the data output will be the bit at the corresponding position within the word stored at the given address.

Distinguishing each memory location in this direct verification requires $n \times m$ unique Boolean variables. There are a further $l = \log_2 n$ Boolean variables for the address input. For even a small memory, the number of variables used and the complexity of the state obtained during simulation are too large for symbolic verification. But with symbolic indexing we can get away with only m variables, $\vec{d} = d_1, \ldots, d_m$ say, to represent the data in *only the addressed* memory location. We write the trajectory assertion as follows:

$$\begin{array}{l} \mathsf{addr\ is}\ \vec{a}\ \ \mathsf{and} \\ (\vec{a} \sim 1 \rightarrow \mathsf{mem}_1\ \mathsf{is}\ \vec{d})\ \mathsf{and} \cdots \mathsf{and}\ (\vec{a} \sim n \rightarrow \mathsf{mem}_n\ \mathsf{is}\ \vec{d}) \end{array} \quad \Rightarrow \quad \mathsf{out\ is}\ \vec{d}.$$

Here, the antecedent assumes that the memory node storing the jth bit of the ith location has the Boolean value d_j under a guard $\vec{a} \sim i$ stating that the ith row is in fact addressed. For example, when the address bits select location 1, the memory node $\mathsf{mem}[1][j]$ is set to d_j, for $1 \leq j \leq m$. But when the address is not 1, these nodes are set to the unknown value X. (It is important to recall that all cases are simulated simultaneously in STE.) Now, with this indexed arrangement, we can expect the jth output node always to have the same Boolean value d_j whatever the value of the address bits, which is what the consequent stipulates. In this verification by symbolic indexing, only $m + l$ variables are required—a very significant reduction. Moreover, in a BDD-based verification, the BDDs representing the bits on each column in the memory array will have a significant amount of sharing, since each of these bits is essentially driven by the same Boolean variable but under different guard conditions. The result is an extremely efficient memory verification computation.

Symbolic indexing is highly effective and widely used in memory verification and similar circuits such as CAMs, but has seen relatively limited adoption for other circuit structures. This is because users have to create the right indexed family of abstractions by manually encoding them into the antecedent. To make symbolic indexing easier, Melham and Jones [47] describe an algorithm for computing a trajectory assertion that encodes a user-given symbolic-indexing scheme from a trajectory assertion that specifies a direct symbolic simulation of the circuit. The transformation

is sound, in the sense that if the resulting trajectory assertion holds then so does the original one. This method, however, still requires the user to specify the indexing scheme, albeit with much less effort than manually writing it into the antecedent.

An algorithm that leverages this transformation to automatically abstract properties by symbolic indexing is presented in [5]. This takes as input a specification for the circuit to be verified, in the form of a Boolean expression that states the required I/O function for the circuit. By analyzing the information requirements for intermediate computations in the specification, the algorithm computes a candidate scheme for symbolic indexing that can then be applied to the verification property using the Melham and Jones transformation. Experimental results show that this approach not only simplifies memory verification, but also enables symbolic indexing of certain other designs fully automatically.

Another technique for memory verification that employs a very different form of symbolic indexing was introduced by a company called InnoLogic Systems, which was later acquired by Synopsys [57]. The technique, described in [78], uses symbolic Boolean variables to encode regular circuit structures, so that a single simulation 'node' records the state values for multiple identical circuit nodes. This enables the verification of very large regular memory arrays, since this encoding overcomes the need to have state information proportional to the number of circuit nodes.

25.8 Compositional Reasoning

The practical application of symbolic trajectory evaluation—indeed of any model-checking method—to complex, industrial-scale circuit designs inevitably faces the serious challenge of fundamental capacity limitations. STE's native abstraction mechanism helps with scalability, and in particular can make the method scale well for circuits whose semantics admits of a good symbolic-indexing scheme. But for many industrial-scale design verifications, abstraction must be complemented by some form of decomposition into sub-problems that STE model checking can handle. To fit into the capacity of the model checker, a high-level correctness property may have to be broken down into many individual trajectory assertions, which combine in potentially complex ways to establish the overall correctness result [51, 65].

To provide a theoretical foundation for problem decomposition, STE can be equipped with a system of formal inference rules for proving trajectory assertions from an axiomatic base of assertions generated by STE model checking [31, 79]. In Sect. 25.8.1, we give a brief overview of one formulation of such a system. We then sketch, in Sect. 25.8.2, an alternative approach to compositional reasoning, called 'Relational STE' [51]. This bypasses the language of trajectory assertions entirely, raising verification properties to a purely logical level suitable for compositional reasoning in ordinary predicate logic.

The use of mechanised, deductive theorem proving—as exemplified by the HOL system [27]—has often been proposed to support compositional reasoning using

systems of STE inference rules. The combination of algorithmic STE model checking and deductive proof was pioneered in the early 1990s in an academic predecessor of Intel's Forte system called Voss [62]. This was followed by numerous experiments in designing and using systems that linked STE and theorem proving of various kinds [2, 4, 31, 39, 61, 65], culminating in a mature integration within Forte of a comparatively full-featured theorem prover, called 'Goaled'. This provides an integrated combination of STE and theorem proving that is seeing increasing use in production verification projects at Intel [51].

25.8.1 The STE Deductive System

In this section, we briefly sketch a system of STE inference rules originally presented by Hazelhurst and Seger in [31]. Another formulation can be found in Hazelhurst's Ph.D. dissertation [30] and is also presented and illustrated by examples in the tutorial [32]. The main use of these deductive systems is to combine individual STE model-checking results together to derive correctness properties that are infeasible to check directly. But the rules can also be used to transform correctness assertions to increase STE model-checking efficiency [2].

Rules of Consequence. These rules are analogous to the classical Hoare logic rules for pre-condition strengthening and post-condition weakening, and are used in proofs for a similar purpose: aligning antecedents and consequents to enable further deductions, primarily transitivity. They are defined semantically, via the ordering on defining sequences, rather than in terms of syntactic implication:

Antecedent Strengthening. For any trajectory formulas A and C, if $\models A \Rightarrow C$ then for any trajectory formula A' for which $[A]^\phi \sqsubseteq [A']^\phi$ for all valuations ϕ, we have $\models A' \Rightarrow C$.
Consequent Weakening. For any trajectory formulas A and C, if $\models A \Rightarrow C$ then for any trajectory formula C' for which $[C']^\phi \sqsubseteq [C]^\phi$ for all valuations ϕ, we have $\models A \Rightarrow C'$.

The Antecedent Strengthening rule says that the antecedent of any proved trajectory assertion can be replaced by one that adds further information about node values, i.e. that has 0 or 1 in place of some Xs, under some valuations of the Boolean variables in the guards. In essence, this rule encapsulates the fundamental abstraction mechanism of STE: complexity is controlled by abstracting away from irrelevant information about node values, and running the STE model-checking algorithm with the weakest possible antecedent that still establishes the consequent. Similarly, Consequent Weakening says that the consequent of a proved trajectory assertion can be replaced by one with less information about node values—we can discard established information about node values.

Logical and Structural Rules. The first of these is a simple axiom:

Reflexivity. $\models A \Rightarrow A$ holds for any trajectory formula A,

that provides a syntactic starting point for deductive proofs. Of course the other way to begin a deduction is to establish one or more trajectory assertions semantically, by STE model checking.

The next two inference rules allow compositional reasoning about complex circuit behaviours, in which 'larger' circuit properties are built up from 'smaller' ones:

Conjunction. For any trajectory formulas A_1, A_2, C_1, and C_2, if $\models A_1 \Rightarrow C_1$ and $\models A_2 \Rightarrow C_2$, then $\models A_1 \text{ and } A_2 \Rightarrow C_1 \text{ and } C_2$.
Transitivity. For any trajectory formulas A, B, and C, if $\models A \Rightarrow B$ and $\models B \Rightarrow C$, then $\models A \Rightarrow C$.

Conjunction allows circuit properties proved separately, typically because of model-checking capacity limits—or because different parts of a large circuit must be simulated under different BDD orderings—to be combined into a single property. If the assumptions expressed by both antecedents hold, then the conjunction of the consequents is established. Transitivity allows one to break a simulation of circuit behaviour over a long period of time into a succession of smaller intervals. Starting from the antecedent A, first establish some relationship B among the values on some intermediate circuit nodes by proving $\models A \Rightarrow B$. Then show that simulation beginning with the assumption that B holds establishes the final consequent C.

Time Shift. This rule allows the baseline time at which simulation of the circuit begins in its initial state to be shifted forward. This is used, for example, to align time points to allow Transitivity to be applied. It also allows a single, general assertion to be instantiated for use in many contexts, differing only in the specific finite interval of time over which the instantiated property needs to specify behaviour. The inference rule is the following:

Time Shift. For any trajectory formulas A and C, if $\models A \Rightarrow C$ then $\models \mathsf{N}\, A \Rightarrow \mathsf{N}\, C$.

Time Shift holds because, in STE, the initial state at which simulation begins is arbitrary. The only constraints on it are those explicitly imposed by the antecedent, as discussed in Sect. 25.5.

Substitution. The final rule is the only one (in this formulation) that has specifically to do with Boolean variables and guards. In simplified form, the rule is:

Substitution. For any trajectory formulas A and C, if $\models A \Rightarrow C$, then $\models A[\vec{P}/\vec{x}] \Rightarrow C[\vec{P}/\vec{x}]$ for any substitution of formulas $\vec{P}$ for Boolean variables $\vec{x}$.

A somewhat more flexible form of the rule, which deals with some essentially syntactic complexities, is presented in [31]. A primary use of Substitution is, again, to align antecedents and consequents so Transitivity applies. Suppose, for example, we have proved some unit-delay input-output relationship $\models \mathsf{a}\ \mathsf{is}\ x \Rightarrow \mathsf{N}(\mathsf{b}\ \mathsf{is}\ E_1)$. Now suppose we separately simulate the next stage in the circuit's sequential behaviour, setting node b to a fresh variable y and proving $\models \mathsf{b}\ \mathsf{is}\ y \Rightarrow \mathsf{N}(\mathsf{c}\ \mathsf{is}\ E_2)$. Using time-shifting, we obtain $\models \mathsf{N}(\mathsf{b}\ \mathsf{is}\ y) \Rightarrow \mathsf{N}(\mathsf{N}(\mathsf{c}\ \mathsf{is}\ E_2))$. We can then use Substitution to

replace the input variable y for the second stage by the output expression E_1 actually computed in the first stage, to obtain $\models \mathsf{N}(\mathsf{b}\ \mathsf{is}\ E_1) \Rightarrow \mathsf{N}(\mathsf{N}(\mathsf{c}\ \mathsf{is}\ E_2[E_1/y]))$. The two stages can then be composed using Transitivity.

As already mentioned, several software tools that combine STE and deductive theorem proving have been designed to support compositional reasoning using this and other systems of specialised STE inference rules. These commonly embed reasoning about STE properties within a more general, higher-order logic: a formalised syntax is introduced that constitutes a 'deep embedding' of trajectory formulas and assertions, and then the inference rules are added to give an axiomatic theory of this embedded 'STE logic'. The connection to model checking is achieved by an axiom scheme that admits any trajectory assertion that has been (externally) checked by the STE model-checking algorithm.

This approach yields a two-level integration, in which the deductive system for inferring circuit properties is largely separate from the general higher-order logic of the theorem prover. Some bridges between the two levels can, however, be achieved by adding certain quantifier rules and axioms about parametric encoding of assumptions. See Sect. VII of [65] for details.

25.8.2 Relational STE

The primitive language of STE trajectory assertions, introduced in Sect. 25.4.1, in essence requires a circuit specification to be functional. Given some inputs, the specification stipulates the values that the outputs shall have, potentially under some constraints that the inputs must satisfy. But many informal circuit specifications seen in practice do not fall into this category—the simplest example being 'nodes a and b are mutually exclusive'. Such *relational* specifications become especially important at higher levels of abstraction, where it may be most appropriate for specifications to be partial [51]. For example, a natural specification of a processor's micro-operation scheduler might say 'a micro-operation with ready sources will be scheduled for execution', while intentionally leaving open the selection between different ready micro-operations. Even at lower abstraction levels, the most natural form of specification may not be functional.

Relational STE is an approach to compositional reasoning about circuit properties that retains STE's underlying symbolic simulation engine but lifts the properties to the level of ordinary predicate logic [51]. This allows much more general properties to be expressed—in particular, properties can be relations, rather than only I/O functions. Perhaps equally important, it also avoids the rather intricate, low-level process of reasoning with specialised STE inference rules illustrated in Sect. 25.8.1. Developed as an extension to Intel's Forte system, Relational STE has been used very effectively for difficult, industrial-scale verifications of floating-point dividers [42] and control-dominated microarchitectural algorithms, such as bus recycle logic and register renaming [40]. These are all circuits for which the natural form of specification is relational.

We now briefly sketch Relational STE, first establishing some basic terminology. Given a circuit c with nodes $\mathscr{N}$, an *execution* is a function $e \in (\mathscr{N} \times \mathbb{N}) \to \mathbb{B}$ that assigns a Boolean value to each circuit node at each point in time. (So an execution is just an STE sequence that is Boolean-valued; see Sect. 25.3.3.) The *behaviour* of a circuit is the set of all its possible executions. We write

$$\|c\| \subseteq (\mathscr{N} \times \mathbb{N}) \to \mathbb{B}$$

for the behaviour of a circuit c. This is simply the classical 'relational' approach to representing circuit behaviour mathematically that is well known from hardware modelling in higher-order logic [17, 46].

Relational STE uses the symbolic simulation algorithm at the heart of STE to check that the behaviour of a circuit, in the sense just defined, satisfies specifications formulated as *constraints*. In essence, a constraint

$$p \subseteq (\mathscr{N} \times \mathbb{N}) \to \mathbb{B}$$

is a predicate on circuit executions that requires a certain relationship to hold among the values that appear on a stipulated set of circuit nodes at some individually specified points of time. The *signature* of a constraint is a finite subset of $\mathscr{N} \times \mathbb{N}$ that defines what these nodes and times are.

Constraints define relationships that are expected to hold among the values on some circuit nodes at certain times. Consider, for example, the informal specification mentioned earlier: 'circuit nodes a and b are mutually exclusive at time point 2'. This can be expressed by the constraint

$$\left\{ e \in (\mathscr{N} \times \mathbb{N}) \to \mathbb{B} \;\middle|\; \overline{e(\mathsf{a}, 2) \wedge e(\mathsf{b}, 2)} \right\}$$

with signature $\{(\mathsf{a}, 2), (\mathsf{b}, 2)\} \subseteq \mathscr{N} \times \mathbb{N}$. This represents a predicate that holds of just those circuit executions in which nodes a and b are mutually exclusive at time 2. Note that this form of specification is expressed—in essence at least—through ordinary (higher-order) propositional logic. For concise presentation, we have explained constraints in set theoretic notation. But it's not hard to see this as a higher-order predicate on functions:

$$\text{mutex}\ e \triangleq \overline{e(\mathsf{a}, 2) \wedge e(\mathsf{b}, 2)}$$

A theorem of higher-order logic that says that all the executions of a circuit c satisfy this predicate would then look something like $\vdash \forall e : (\mathscr{N} \times \mathbb{N}) \to \mathbb{B} \,.\, e \in \|c\| \supset \text{mutex}\ e$.

In practice, circuit correctness properties are formulated in Relational STE as a pair of constraints: an *input constraint*, P_i, and an *output constraint*, P_o. Given such a pair, the underlying symbolic simulation algorithm of STE is used to check that any circuit execution that satisfies the input constraint will (under simulation) also satisfy the output constraint. Formally, Relational STE proves correctness statements of the form $\vdash \forall e : (\mathscr{N} \times \mathbb{N}) \to \mathbb{B} \,.\, e \in \|c\| \supset (\mathrm{P}_i\ e \supset \mathrm{P}_o\ e)$.

The algorithm for verifying these relational correctness properties is simple, at least conceptually: it is just Boolean symbolic simulation. First construct a classical STE antecedent that attaches a unique, unconstrained Boolean variable to every circuit node. Then run the STE simulator—for as many steps as needed, according to the signatures of the input and output constraints—and record all the symbolic node values that arise at each step of the simulation. This produces a single, *symbolic* execution, $\hat{e}$ say, that encompasses all the behaviours of the circuit. The implication $\mathrm{P}_i\ \hat{e} \supset \mathrm{P}_o\ \hat{e}$ can then be evaluated to get the result using BDDs or a SAT solver.

Of course this simplistic algorithm is vastly too complex, computationally, to be practical for industrial-scale problems. The implementation of Relational STE in Intel's Forte system employs a sophisticated array of optimisations that exploit the power of native STE. For example, not all nodes are set to variables at the start of simulation, only the ones needed to evaluate the input and output constraints. (Or that have to be Boolean to avoid weak disagreements; see Sect. 25.6.) The remaining inputs have value X at the start of the simulation, raising the general level of abstraction and lowering complexity. In addition, many inputs are typically set to constants, for example clock patterns and certain testability signals. More importantly, some elements of the input constraint will be injected into the simulation using parametric representation, as discussed in Sect. 25.6.3. Dynamic weakening and some other abstraction mechanisms are also used. Exploitation of the full power of symbolic indexing, however, remains for future development of the framework [51].

This relational form of symbolic trajectory evaluation preserves the power of the underlying STE algorithm while enabling much richer specifications: ones that stipulate relations among node values, rather than just functions. Relational STE also eliminates the need to use specialised STE inference rules and apparatus for temporal reasoning; the relational formulation makes it 'just' higher-order logic. Since its introduction, Relational STE has been the workhorse of data-path formal verification at Intel; many thousands of individual operations have been verified in several microprocessor families and over the course of several generations [51].

25.9 GSTE and Other Extensions

Relational STE essentially discards the fixed temporal logic of STE trajectory logic in favour of much less restricted assertions about behaviour under simulation, expressed in conventional predicate logic. This represents a step away from the classical idea of 'model checking' temporal formulas. Almost since the inception of STE, however, variants and extensions of STE have been formulated that remain in the realm of temporal model checking, but have more expressive specification languages than the simple language of trajectory formulas and trajectory assertions introduced in Sect. 25.4.

In their cornerstone article on STE [64], Seger and Bryant formulate a version with higher expressive power at the level of trajectory *assertions*. In addition to the basic form of trajectory assertion, $A \Rightarrow C$, they allow sequences $A \Rightarrow C\ ;\ G$ and

iterations $(A \Rightarrow C)^\star; G$, where G is another trajectory assertion. Sequences allow circuit behaviours to be specified as the concatenation of a temporal succession of sub-behaviours. Roughly speaking, an iteration $(A \Rightarrow C)^\star; G$ specifies that any trajectory of the circuit must satisfy C for as long as it satisfies A, after which it will satisfy G. Neither form of correctness assertion is reported to have found widespread use in industrial practice.

In their comprehensive STE tutorial [32], Hazelhurst and Seger give a version of STE in which the property language, called TL, is a four-valued linear-time temporal logic that includes negation and a strong Until operator. In contrast to the trajectory formulas of Sect. 25.4, the formulas of TL have four possible *truth-values*, true (**t**), false (**f**), unknown ($\bot$), and inconsistent ($\top$), ordered in a lattice in the obvious way. It is important to distinguish between this four-valued lattice of *truth-values* of TL formulas and the lattice of *state values*, ordered by information content, introduced in this chapter in Sect. 25.3.1. The former represents the *degree of knowledge* we have of the truth or falsity of propositions in the property language; the latter captures the *amount of information* there is about the values on circuit nodes. Hazelhurst and Seger give an interesting justification for maintaining this distinction [32, p. 19], a key point of which is that it allows a formulation of STE with negation.

The theory of STE in this extended form is a little more involved than the simpler version presented in this chapter. For example, there is no longer a single 'defining' sequence and trajectory for a formula—instead TL formulas have defining *sets* of minimal sequences and trajectories. An analogue of the Fundamental Theorem of Trajectory Evaluation (see Sect. 25.5) can be obtained in this system, but exploiting this to actually check properties in an algorithmic way is rather more involved than the method sketched in Sect. 25.6. In practice, certain restrictions must be placed on the formulas checked, and some approximations are involved. Full details of the theory and model-checking algorithms—as well as a theory of compositional reasoning—can be found in Hazelhurst's Ph.D. dissertation [30].

This chapter has focussed on *bit-level* hardware models and their verification in symbolic trajectory evaluation—a level of hardware modelling at which STE has been spectacularly successful for industrial data-path verification. Right from the start, however, the theory of STE was framed in the much more general setting of an arbitrary lattice of state abstractions [64], and was not restricted to the bit-level abstraction of Boolean states introduced in Sect. 25.3.2. The idea was that one could have STE-style algorithms that analyse systems at a higher (or at least different) level of *data abstraction* [46], provided a good representation for the indexed families of abstract states could be devised. The arrival of highly efficient SMT solvers has paved the way for this ambition to be realised in the form of *word-level STE*. This is a verification method based on symbolic simulation and a lattice of abstract states in which the underlying notion of concrete data is a group of binary digits, rather than individual bits. Work is underway at Intel to create such word-level STE verification tools [63].

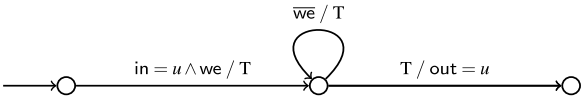


Fig. 8 Memory cell specification as an assertion graph

25.9.1 Generalized Symbolic Trajectory Evaluation

A more radical departure from classical STE is Generalized Symbolic Trajectory Evaluation, GSTE. This aims to preserve the power of abstraction and efficiency benefits of STE while making the property language much more expressive, using some of the fundamental techniques of classical symbolic model checking.

GSTE builds on several prior efforts to increase the expressive power of STE, including the introduction of iterations in [64]. Beatty first generalised specifications to arbitrary labelled transition graphs [7], and a model-checking algorithm for these specifications was proposed by Nelson and Jain [48]. Jain proposed a semantics that existentially quantifies over paths in a transition graph and gave an incomplete model-checking algorithm [35]. The next step was the insightful development by Chou of a more natural semantics that universally quantifies over paths, together with a sound and complete model-checking algorithm [19]. Yang and Seger then introduced GSTE, using a graphical specification notation, and adding backwards reasoning and other algorithmic developments [76, 77]. GSTE has also been further extended to introduce a variant called *concurrent* (or *compositional*) GSTE [72, 75]. This allows different parts of circuits to be analysed by independent simulations and the results combined.

GSTE overcomes the two main limitations of classical STE. It can express properties over *unbounded* periods of time, and it allows reasoning by propagation of information *backward* in time. The resulting extension is significant—algorithms for GSTE can verify all ω-regular properties [77]. Specifications are presented in a notation called *assertion graphs*. These are directed graphs with an initial vertex, where each edge is labelled by *antecedent* and *consequent* conditions on circuit nodes. The vertexes represent sets of circuit states and the edges discrete transitions between them. As with conventional STE, the antecedents drive simulation with a constrained input stimulus and the consequents stipulate the expected circuit response. Roughly speaking, the property specified by an assertion graph is the following. The graph defines a set of finite paths, each starting at the initial vertex. For each such path, all finite sequences of circuit states of the same length that satisfy all the antecedents along the path must also satisfy all the consequents along the path.

A simple example, taken from [68], is the memory cell specification in Fig. 8. The edges are labelled with pairs of the form *antecedent* / *consequent*. The property specified is that whenever the write enable node we is high and the input node has the value named symbolically by u, the output node out should subsequently present the value u for as long as no further writes are enabled. Much more complex examples

can be found in the literature. The verification of FIFO circuits is explored in [74], and a FIFO case study is again used to illustrate GSTE in [73]. The specification and verification of a complex memory unit is explained in [77]. Concurrent GSTE is applied to the verification of Intel Pentium 4 scheduler circuits in [72, 75].

The GSTE model-checking algorithm presented in [77] records a set of states of the circuit model for each assertion graph edge. This contains all the states that are reachable along some path from the initial vertex via a trace of circuit execution that satisfies all the antecedents along the path. The algorithm is similar to reachability analysis, computing the fixed point of taking the post-images of each transition. This and other connections between GSTE and conventional symbolic model checking are discussed in [59]. Overlaid onto this basic algorithmic framework are a number of technical devices for controlling and localising abstraction levels across an assertion graph. These include a mechanism for limiting the scope of symbolic variables to a specific edge [74] and for manually controlling certain other points at which variables are quantified—so called *knots* [49]. Another variable-handling mechanism, called *precise nodes* [74], is used to maintain node value inter-dependencies across temporally overlapping scopes.

As discussed in Sect. 25.4.1, an essential limitation of conventional formulations of STE is that they reason by forwards simulation only. They therefore cannot verify properties that require the propagation of information backwards along state transitions. GSTE aims to overcome this by adding an initial phase to the model-checking algorithm; a pre-image fixed-point calculation is done that strengthens earlier antecedent constraints in the assertion graph by propagating later antecedent constraints backwards. See [74, 77] for more details of this algorithm.

Although Generalized Symbolic Trajectory Evaluation has seen some success at Intel Corporation, it has not—at the time of writing—yet been established in widespread use. Moreover, as some of the practical complexities just hinted at suggest, GSTE may not yet be mature and may see further development and modification. For example, Smith's Ph.D. dissertation recasts GSTE into a system comprising *generalized trajectory logic*, a low-level temporal logic for reasoning about symbolic ternary simulations, and *assertion programs*, an executable high-level specification language [68]. The resulting system is, it is claimed, cleaner and more amenable to formal reasoning. We therefore do not give full technical details of GSTE in this chapter, but refer the reader to the literature cited in this section. Good starting points for learning about GSTE in depth are the background section of Smith's Ph.D. dissertation [68, Sect. 2.5] and the semantic account by Claessen and Roorda [21].

25.10 Summary and Prospects

STE and its descendent GSTE provide a uniquely effective approach to formal verification of difficult, industrial-scale circuit designs—especially data-paths, but some control-dominated designs too. Their effectiveness comes from a combination of

symbolic simulation with a two-level system of abstraction, whereby (overlapping) families of abstractions can be represented compactly and checked simultaneously. Intel Corporation has been a prominent user, and developer, of this model-checking technology. At the time of writing, Intel's deployment of STE through its Forte environment was one of the most substantial and sustained formal (property) verification efforts anywhere in industry. But STE-based formal verification has also seen significant use at Motorola and IBM.

More recently, an in-house framework for processor verification has been developed at Centaur Technology that has many parallels with Intel's Forte environment, as well as some significant differences [34, 66]. This, too, is based on symbolic circuit simulation, provides abstraction, and has a logic for compositional reasoning. A notable feature of the Centaur framework is that it is built on top of publicly available software tools: the well-established ACL2 [43] theorem prover and special-purpose tools such as the ZZ framework [25] and ABC [9].

The successful industrial deployment of two major verification frameworks based on symbolic simulation—Forte at Intel and the ACL2-based tools at Centaur Technology—suggest that this idea has come of age industrially, at least for processor verification. Moreover the parallels between the two systems, each quite different from the other in numerous matters of detail, strengthens the conclusion that this *kind* of approach represents a general solution in this important domain. To date, there are no commercial STE tools, though the technology is well documented and seems ripe for more widespread take-up.

GSTE is not as well established as STE, but seems to be a very promising idea that merits much further development and experimentation. Perhaps the best way to think of GSTE, with its assertion graphs or (in Smith's formulation) assertion programs, is as a framework for articulating 'reference' hardware algorithms at a flexible, intermediate layer of abstraction above the circuit level.

There is, of course, a very sizable literature on partitioning and abstraction in both hardware and software verification. A few connections have been made in the literature between the use of these ideas in other settings and the specific mechanisms provided in STE and GSTE. For example, a combination of partitioned abstraction, similar to that provided in GSTE and STE, with predicate abstraction and counterexample-guided abstraction refinement in the context of symbolic model checking, is explored in [60]. A fruitful research direction would be to look for other ways in which underlying ideas of STE and GSTE, which have made them so successful in processor verification, might be adapted to other contexts, especially software verification.

Acknowledgements Carl Seger and Randy Bryant are the originators of STE, and GSTE is due to Jin Yang and Carl Seger. I am grateful for many illuminating discussions of STE with Carl, and extremely grateful for an extended and close collaboration with Carl and many other Intel scientists and engineers who have contributed to STE and its embodiment in the Forte system. Ed Smith's Ph.D. dissertation provided a fresh perspective on GSTE. John Harrison, my students at Oxford, and anonymous referees all provided insightful comments that improved the presentation.

References

1. Aagaard, M.D., Jones, R.B., Melham, T.F., O'Leary, J.W., Seger, C.J.H.: A methodology for large-scale hardware verification. In: Hunt, W.A. Jr., Johnson, S.D. (eds.) Formal Methods in Computer Aided Design (FMCAD). LNCS, vol. 1954, pp. 263–282. Springer, Heidelberg (2000)
2. Aagaard, M.D., Jones, R.B., Seger, C.J.H.: Combining theorem proving and trajectory evaluation in an industrial environment. In: Design Automation Conf. (DAC), pp. 538–541. IEEE, Piscataway (1998)
3. Aagaard, M.D., Jones, R.B., Seger, C.J.H.: Formal verification using parametric representations of Boolean constraints. In: Design Automation Conf. (DAC), pp. 402–407. ACM, New York (1999)
4. Aagaard, M.D., Jones, R.B., Seger, C.J.H.: Lifted-FL: a pragmatic implementation of combined model checking and theorem proving. In: Bertot, Y., Dowek, G., Hirschowitz, A., Paulin, C., Théry, L. (eds.) Theorem Proving in Higher Order Logics (TPHOLs). LNCS, vol. 1690, pp. 323–340. Springer, Heidelberg (1999)
5. Adams, S., Björk, M., Melham, T., Seger, C.J.: Automatic abstraction in symbolic trajectory evaluation. In: Baumgartner, J., Sheeran, M. (eds.) Formal Methods in Computer Aided Design (FMCAD), pp. 127–135. IEEE, Piscataway (2007)
6. Ball, T., Podelski, A., Rajamani, S.K.: Boolean and Cartesian abstraction for model checking C programs. Int. J. Softw. Tools Technol. Transf. **5**, 49–58 (2003)
7. Beatty, D.L., Bryant, R.E.: Formally verifying a microprocessor using a simulation methodology. In: Design Automation Conf. (DAC), pp. 596–602. ACM, New York (1994)
8. Beer, I., Ben-David, S., Eisner, C., Rodeh, Y.: Efficient detection of vacuity in ACTL formulas. Form. Methods Syst. Des. **18**(2), 141–162 (2001)
9. Berkeley Logic Synthesis and Verification Group: ABC: a system for sequential synthesis and verification. http://www.eecs.berkeley.edu/~alanmi/abc/
10. Bhadra, J., Martin, A., Abraham, J., Abadir, M.: A language formalism for verification of PowerPC custom memories using compositions of abstract specifications. In: High-Level Design Validation and Test Workshop, pp. 134–141. IEEE, Piscataway (2001)
11. Bhadra, J., Martin, A.K., Abraham, J.A., Abadir, M.S.: Using abstract specifications to verify PowerPC custom memories by symbolic trajectory evaluation. In: Correct Hardware Design and Verification Methods (CHARME). LNCS, vol. 1690, pp. 386–402. Springer, Heidelberg (2001)
12. Bird, R.: Introduction to Functional Programming using Haskell, 2nd edn. Prentice Hall, New York (1998)
13. Bryant, R.E.: Formal verification of memory circuits by switch-level simulation. Trans. Comput.-Aided Des. Integr. Circuits Syst. **10**(1), 94–102 (1991)
14. Bryant, R.E.: A methodology for hardware verification based on logic simulation. J. ACM **38**(2), 299–328 (1991)
15. Bryant, R.E., Beatty, D., Brace, K., Cho, K., Sheffler, T.: COSMOS: a compiled simulator for MOS circuits. In: Design Automation Conf. (DAC), pp. 9–16. ACM, New York (1987)
16. Bryant, R.E., Beatty, D.E., Seger, C.J.H.: Formal hardware verification by symbolic ternary trajectory evaluation. In: Design Automation Conf. (DAC), pp. 297–402. IEEE, Piscataway (1991)
17. Camilleri, A., Gordon, M., Melham, T.: Hardware verification using higher-order logic. In: Borrione, D. (ed.) From HDL Descriptions to Guaranteed Correct Circuit Designs, pp. 43–67. North-Holland, Amsterdam (1987)
18. Chockler, H., Grumberg, O., Yadgar, A.: Efficient automatic STE refinement using responsibility. In: Ramakrishnan, R., Rehof, J. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 4963, pp. 233–248. Springer, Heidelberg (2008)

19. Chou, C.T.: The mathematical foundation of symbolic trajectory evaluation. In: Halbwachs, N., Peled, D. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 1633, pp. 196–207. Springer, Heidelberg (1999)
20. Claessen, K., Roorda, J.W.: An introduction to symbolic trajectory evaluation. In: International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM). LNCS, vol. 3965, pp. 56–77. Springer, Heidelberg (2006)
21. Claessen, K., Roorda, J.W.: A faithful semantics for generalised symbolic trajectory evaluation. Log. Methods Comput. Sci. **5**(2), 1–32 (2009)
22. Clarke, E.M., Grumberg, O., Long, D.E.: Model checking and abstraction. In: Symp. on Principles of Programming Languages (POPL), pp. 343–354. ACM, New York (1992)
23. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
24. Davey, B.A., Priestley, H.A.: Introduction to Lattices and Order. Cambridge University Press, Cambridge (1990)
25. Een, N.: ABC/ZZ. https://bitbucket.org/niklaseen/abc-zz
26. Flaisher, A., Gluska, A., Singerman, E.: Case study: integrating FV and DV in the verification of the Intel Core 2 Duo microprocessor. In: Baumgartner, J., Sheeran, M. (eds.) Formal Methods in Computer Aided Design (FMCAD), pp. 192–195. IEEE, Piscataway (2007)
27. Gordon, M.J.C., Melham, T.F. (eds.): Introduction to HOL: A Theorem Proving Environment for Higher Order Logic. Cambridge University Press, Cambridge (1993)
28. Grumberg, O.: Abstraction and refinement in model checking. In: de Boer, F., Bonsangue, M., Graf, S., de Roever, W.P. (eds.) Formal Methods for Components and Objects. LNCS, vol. 4111, pp. 219–242. Springer, Heidelberg (2006)
29. Halmos, P.R.: Naive Set Theory. Springer, Heidelberg (1987)
30. Hazelhurst, S.: Compositional model checking of partially ordered state spaces. Technical report 96-02, Department of Computer Science, University of British, Columbia (1996)
31. Hazelhurst, S., Seger, C.J.H.: A simple theorem prover based on symbolic trajectory evaluation and BDDs. Trans. Comput.-Aided Des. Integr. Circuits Syst. **14**(4), 413–422 (1995)
32. Hazelhurst, S., Seger, C.J.H.: Symbolic trajectory evaluation. In: Kropf, T. (ed.) Formal Hardware Verification, pp. 3–78. Springer, Heidelberg (1997). Chap. 1
33. Hellerman, L.: A catalog of three-variable or-invert and and-invert logical circuits. Trans. Electron. Comput. **EC-12**(3), 198–223 (1963)
34. Hunt, W.A. Jr., Swords, S., Davis, J., Slobodova, A.: Use of formal verification at Centaur Technology. In: Hardin, D.S. (ed.) Design and Verification of Microprocessor Systems for High-Assurance Applications, pp. 65–88. Springer, Heidelberg (2010)
35. Jain, A.: Formal hardware verification by symbolic trajectory evaluation. Ph.D. thesis, Carnegie Mellon University (1997)
36. Jain, P., Gopalakrishnan, G.: Efficient symbolic simulation-based verification using the parametric form of Boolean expressions. Trans. Comput.-Aided Des. Integr. Circuits Syst. **13**(11), 1005–1015 (1994)
37. Jones, R.B.: Applications of symbolic simulation to the formal verification of microprocessors. Ph.D. thesis, Department of Electrical Engineering, Stanford University (1999)
38. Jones, R.B., O'Leary, J.W., Seger, C.J.H., Aagaard, M.D., Melham, T.F.: Practical formal verification in microprocessor design. IEEE Des. Test Comput. **18**(4), 16–25 (2001)
39. Joyce, J., Seger, C.J.: Linking BDD-based symbolic evaluation to interactive theorem-proving. In: Design Automation Conf. (DAC), pp. 469–474. IEEE, Piscataway (1993)
40. Kaivola, R.: Formal verification of Pentium® 4 components with symbolic simulation and inductive invariants. In: Etessami, K., Rajamani, S.K. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 3576, pp. 170–184. Springer, Heidelberg (2005)
41. Kaivola, R., Ghughal, R., Narasimhan, N., Telfer, A., Whittemore, J., Pandav, S., Slobodova, A., Taylor, C., Frolov, V., Reeber, E., Naik, A.: Replacing testing with formal verification in Intel Core i7 processor execution engine validation. In: Bouajjani, A., Maler, O. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, pp. 414–429. Springer, Heidelberg (2009)

42. Kaivola, R., Kohatsu, K.R.: Proof engineering in the large: formal verification of Pentium 4 floating-point divider. Int. J. Softw. Tools Technol. Transf. **4**(3), 323–334 (2003)
43. Kaufmann, M., Moore, J.S.: An industrial strength theorem prover for a logic based on common Lisp. Trans. Softw. Eng. **23**(4), 203–213 (1997)
44. Khasidashvili, Z., Gavrielov, G., Melham, T.: Assume-guarantee validation for STE properties within an SVA environment. In: Biere, A., Pixley, C. (eds.) Formal Methods in Computer Aided Design (FMCAD), pp. 108–115. IEEE, Piscataway (2009)
45. Krishnamurthy, N., Martin, A.K., Abadir, M.S., Abraham, J.A.: Validating PowerPC custom memories. IEEE Des. Test Comput. **17**(4), 61–76 (2000)
46. Melham, T.: Higher Order Logic and Hardware Verification. Cambridge University Press, Cambridge (1993)
47. Melham, T.F., Jones, R.B.: Abstraction by symbolic indexing transformations. In: Aagaard, M.D., O'Leary, J.W. (eds.) Formal Methods in Computer Aided Design (FMCAD). LNCS, vol. 2517, pp. 1–18. Springer, Heidelberg (2002)
48. Nelson, K.L., Jain, A., Bryant, R.E.: Formal verification of a superscalar execution unit. In: Design Automation Conf. (DAC), pp. 161–166. ACM, New York (1997)
49. Ng, K., Hu, A.J., Yang, J.: Generating monitor circuits for simulation-friendly GSTE assertion graphs. In: Grochowski, E., Dillinger, T. (eds.) Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD), pp. 409–416. IEEE, Piscataway (2004)
50. Nguyen, M.D., Thalmaier, M., Wedler, M., Bormann, J., Stoffel, D., Kunz, W.: Unbounded protocol compliance verification using interval property checking with invariants. Trans. Comput.-Aided Des. Integr. Circuits Syst. **27**(11), 2068–2082 (2008)
51. O'Leary, J., Kaivola, R., Melham, T.: Relational STE and theorem proving for formal verification of industrial circuit designs. In: Jobstmann, B., Ray, S. (eds.) Formal Methods in Computer Aided Design (FMCAD), pp. 97–104. IEEE, Piscataway (2013)
52. Pandey, M., Raimi, R., Bryant, R.E., Abadir, M.S.: Formal verification of content addressable memories using symbolic trajectory evaluation. In: Design Automation Conf. (DAC), pp. 167–172. ACM, New York (1997)
53. Roorda, J.W.: Semantics, decision procedures, and abstraction refinement for symbolic trajectory evaluation. Ph.D. thesis, Chalmers University of Technology and Göteborg Universty (2006)
54. Roorda, J.W., Claessen, K.: Explaining symbolic trajectory evaluation by giving it a faithful semantics. In: Grigoriev, D., Harrison, J., Hirsch, E.A. (eds.) International Computer Science Symposium in Russia (CSR). LNCS, vol. 3967, pp. 555–566. Springer, Heidelberg (2006)
55. Roorda, J.W., Claessen, K.: SAT-based assistance in abstraction refinement for symbolic trajectory evaluation. In: Intl. Conf. on Computer-Aided Verification (CAV), pp. 175–189. Springer, Heidelberg (2006)
56. Ryan, M., Sadler, M.: Valuation systems and consequent relations. In: Abramsky, S., Gabbay, D.M., Maibaum, T.S.E. (eds.) Handbook of Logic in Computer Science, vol. 1, pp. 1–78. Oxford University Press, Oxford (1992). Chap. 1
57. Santarini, M.: Synopsys extends formal reach with InnoLogic acquisition. EE Times (2003). www.eetimes.com/document.asp?doc_id=1216962
58. Schubert, T.: High level formal verification of next-generation microprocessors. In: Design Automation Conf. (DAC), pp. 1–6. ACM, New York (2003)
59. Sebastiani, R., Singerman, E., Tonetta, S., Vardi, M.Y.: GSTE is partitioned model checking. Form. Methods Syst. Des. **31**(2), 177–196 (2007)
60. Sebastiani, R., Tonetta, S., Vardi, M.: Property-driven partitioning for abstraction refinement. In: Grumberg, O., Huth, M. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 4424, pp. 389–404. Springer, Heidelberg (2007)
61. Seger, C.J., Joyce, J.: A mathematically precise two-level formal hardware verification methodology. Report 92-34, Department of Computer Science, University of British, Columbia (1992)

62. Seger, C.J.H.: Voss—a formal hardware verification system: user's guide. Tech. Rep. TR-93-45, University of British Columbia, Department of Computer Science (1993)
63. Seger, C.J.H.: Personal communication (2014)
64. Seger, C.J.H., Bryant, R.E.: Formal verification by symbolic evaluation of partially-ordered trajectories. Form. Methods Syst. Des. **6**(2), 147–189 (1995)
65. Seger, C.J.H., Jones, R.B., O'Leary, J.W., Melham, T., Aagaard, M.D., Barrett, C., Syme, D.: An industrially effective environment for formal hardware verification. Trans. Comput.-Aided Des. Integr. Circuits Syst. **24**(9), 1381–1405 (2005)
66. Slobodová, A., Davis, J., Swords, S., Hunt, W.: A flexible formal verification framework for industrial scale validation. In: International Conference on Formal Methods and Models for Codesign, pp. 89–97. IEEE, Piscataway (2011)
67. Smith, E.: A logic for GSTE. In: Baumgartner, J., Sheeran, M. (eds.) Formal Methods in Computer Aided Design (FMCAD), pp. 119–126. IEEE, Piscataway (2007)
68. Smith, E.: Specifying properties for generalized symbolic trajectory evaluation. Ph.D. thesis, University of Oxford (2008)
69. Tzoref, R., Grumberg, O.: Automatic refinement and vacuity detection for symbolic trajectory evaluation. In: Ball, T., Jones, R.B. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 4144, pp. 190–204. Springer, Heidelberg (2006)
70. Velev, M.N., Bryant, R.E.: Efficient modeling of memory arrays in symbolic ternary simulation. In: Steffen, B. (ed.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 1384, pp. 136–150. Springer, Heidelberg (1998)
71. Wang, L.C., Abadir, M.S., Krishnamurthy, N.: Automatic generation of assertions for formal verification of PowerPC microprocessor arrays using symbolic trajectory evaluation. In: Design Automation Conf. (DAC), pp. 534–537. IEEE, Piscataway (1998)
72. Yang, J., Ghughal, R., Tiemeyer, A.: Industrial scale formal verification using concurrent GSTE. In: Tang, T.A., Huang, Y. (eds.) International Conference on ASIC (ASICON), pp. 930–933. IEEE, Piscataway (2005)
73. Yang, J., Goel, A.: GSTE through a case study. In: Pileggi, L., Kuelmann, A. (eds.) International Conference on Computer Aided Design (ICCAD), pp. 534–541. ACM, New York (2002)
74. Yang, J., Seger, C.H.: Generalized symbolic trajectory evaluation—abstraction in action. In: Aagaard, M.D., O'Leary, J.W. (eds.) Formal Methods in Computer Aided Design (FMCAD). LNCS, vol. 2517, pp. 70–87. Springer, Heidelberg (2002)
75. Yang, J., Seger, C.J.: Compositional specification and model checking in GSTE. In: Alur, R., Peled, D.A. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 3114, pp. 216–228. Springer, Heidelberg (2004)
76. Yang, J., Seger, C.J.H.: Generalized symbolic trajectory evaluation. Tech. rep., Intel Strategic CAD Labs (2000)
77. Yang, J., Seger, C.J.H.: Introduction to generalized symbolic trajectory evaluation. Trans. Very Large Scale Integr. (VLSI) Syst. **11**(3), 345–353 (2003)
78. Zhong, J.X.: Circuit simulation using encoding of repetitive subcircuits (2001). US Patent Number 6,865,525
79. Zhu, Z., Seger, C.J.: The completeness of a hardware inference system. In: Dill, D.L. (ed.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 818, pp. 286–298. Springer, Heidelberg (1994)

Chapter 26
The mu-calculus and Model Checking

Julian Bradfield and Igor Walukiewicz

Abstract This chapter presents that part of the theory of the μ-calculus that is relevant to the model-checking problem as broadly understood. The μ-calculus is one of the most important logics in model checking. It is a logic with an exceptional balance between expressiveness and algorithmic properties.

The chapter describes at length the game characterization of the semantics of the μ-calculus. It discusses the theory of the μ-calculus starting with the tree-model property, and bisimulation invariance. Then it develops the notion of modal automaton: an automaton-based model behind the μ-calculus. It gives a quite detailed explanation of the satisfiability algorithm, followed by results on alternation hierarchy, proof systems, and interpolation. Finally, the chapter discusses the relation of the μ-calculus to monadic second-order logic as well as to some program and temporal logics. It also presents two extensions of the μ-calculus that allow us to address issues such as inverse modalities.

26.1 Introduction

The μ-calculus is one of the most important logics in model checking. It is a logic with an exceptional balance between expressiveness and algorithmic properties. In this chapter we present that part of the theory of the μ-calculus that seems to us most relevant to the model-checking problem as broadly understood.

This chapter is divided into three parts. In Sect. 26.2 we introduce the logic, and present some basic notions such as: special forms of formulas, vectorial syntax, and alternation depth of fixpoints. The largest part of this section is concerned with a characterization of the semantics of the logic in terms of games. We give a relatively detailed exposition of the characterization, since in our opinion this is one of the central tools in the theory of the μ-calculus. The section ends with an overview of approaches to the model-checking problem for the logic.

J. Bradfield
University of Edinburgh, Edinburgh, UK

I. Walukiewicz (✉)
CNRS, University of Bordeaux, Bordeaux, France
e-mail: igw@labri.fr

E.M. Clarke et al. (eds.), *Handbook of Model Checking*,
DOI 10.1007/978-3-319-10575-8_26

Section 26.3 goes deeper into the theory of the μ-calculus. It starts with the tree-model property, and bisimulation invariance. Then it develops the notion of modal automaton: an automaton-based model behind the μ-calculus. This model is then often used in the rest of the chapter. We continue the section with a quite detailed explanation of the satisfiability algorithm. This is followed by results on alternation hierarchy, proof systems, and interpolation. We finish with a division property that is useful for modular verification and synthesis.

Section 26.4 presents the μ-calculus in a larger context. We relate the logic to monadic second-order logic as well as to some program and temporal logics. We also present two extensions of the μ-calculus that allow us to express inverse modalities, some form of equality, or counting.

This chapter is short, given the material that we would like to cover. Instead of being exhaustive, we try to focus on concepts and ideas we consider important and interesting from the perspective of the model-checking problem as broadly understood. Since concepts often give more insight than enumeration of facts, we give quite complete arguments for the main results we present.

26.2 Basics

In this section we present some basic notions and tools of the theory of the μ-calculus. We discuss some special forms of formulas such as guarded or vectorial forms. We introduce also the notion of alternation depth. Much of this section is devoted to a characterization of the semantics of the logic in terms of parity games, and its use in model checking. The section ends with an overview of model-checking methods and results.

26.2.1 Syntax and Semantics

The μ-calculus is a logic describing properties of transition systems: potentially infinite graphs with labeled edges and vertices. Often the edges are called *transitions* and the vertices *states*. Transitions are labeled with *actions*, $Act = \{a, b, c, \dots\}$, and the states with sets of *propositions*, $Prop = \{p_1, p_2, \dots\}$. Formally, a *transition system* is a tuple:

$$\mathcal{M} = \langle S, \{R_a\}_{a \in Act}, \{P_i\}_{i \in \mathbb{N}} \rangle$$

consisting of a set S of states, a binary relation $R_a \subseteq S \times S$ defining transitions for every action $a \in Act$, and a set $P_i \subseteq S$ for every proposition. A pair $(s, s') \in R_a$ is called an *a-transition*.

We require a countable set of *variables*, whose meanings will be sets of states. These can be bound by fixpoint operators to form fixpoint formulas. We use $Var = \{X, Y, Z \dots\}$ for variables.

Syntax. The formulas of the logic are constructed using conjunction, disjunction, modalities, and fixpoint operators. The set of μ-calculus formulas $\mathcal{F}_{mc}$ is the smallest set containing:

- p and $\neg p$ for all propositions $p \in Prop$;
- X for all variables $X \in Var$;
- $\alpha \vee \beta$ as well as $\alpha \wedge \beta$, if α, β are formulas in $\mathcal{F}_{mc}$;
- $\langle a \rangle \alpha$ and $[a]\alpha$, if $a \in Act$ is an action and α is a formula in $\mathcal{F}_{mc}$;
- $\mu X.\alpha$ and $\nu X.\alpha$, if $X \in Var$ is a variable and $\alpha \in \mathcal{F}_{mc}$ is a formula.

Disambiguating parentheses are added when necessary. It is generally agreed that $\langle a \rangle$ and $[a]$ bind more tightly than Boolean operators, but opinions vary on whether μ and ν bind more or less tightly than Booleans. We will assume that they bind more loosely. For example, consider the important formula "infinitely often p on some path", which fully parenthesized is $\nu Y.(\mu X.((p \wedge \langle a \rangle Y) \vee \langle a \rangle X))$. We shall write it as $\nu Y.\mu X.(p \wedge \langle a \rangle Y) \vee \langle a \rangle X$.

We write $\sigma X.\alpha$ to stand for $\mu X.\alpha$ or $\nu X.\alpha$. We will use tt as an abbreviation of $(p_1 \vee \neg p_1)$, and ff for $(p_1 \wedge \neg p_1)$. Note that there is no negation operation in the syntax; we just permit negations of propositions. Actually, the operation of the negation of a sentence will turn out to be definable.

Semantics. The semantics of the logic is concise and yet of intriguing depth. We will see later that it pays to study it in detail from different points of view. The meaning of a formula in a transition system is the set of states satisfying the formula. Since a formula may have free variables, their meaning should be fixed before evaluating the formula. As with formulas, the meaning of a variable will be a set of states of the transition system. More formally, given a transition system $\mathcal{M} = \langle S, \{R_a\}_{a \in Act}, \{P_i\}_{i \in \mathbb{N}} \rangle$ and a valuation $\mathcal{V} : Var \to \mathcal{P}(S)$ we define the meaning of a formula $[\![\alpha]\!]^{\mathcal{M}}_{\mathcal{V}}$ by induction on its structure. The meaning of variables is given by the valuation. The meanings of propositional constants and their negations are given by the transition system.

$$[\![X]\!]^{\mathcal{M}}_{\mathcal{V}} = \mathcal{V}(X), \quad \text{for every } X \in \mathcal{V};$$

$$[\![p_i]\!]^{\mathcal{M}}_{\mathcal{V}} = P_i \quad \text{and} \quad [\![\neg p_i]\!]^{\mathcal{M}}_{\mathcal{V}} = S - P_i, \quad \text{for every } p_i \in Prop.$$

Disjunction and conjunction are interpreted as union and intersection:

$$[\![\alpha \vee \beta]\!]^{\mathcal{M}}_{\mathcal{V}} = [\![\alpha]\!]^{\mathcal{M}}_{\mathcal{V}} \cup [\![\beta]\!]^{\mathcal{M}}_{\mathcal{V}} \qquad [\![\alpha \wedge \beta]\!]^{\mathcal{M}}_{\mathcal{V}} = [\![\alpha]\!]^{\mathcal{M}}_{\mathcal{V}} \cap [\![\beta]\!]^{\mathcal{M}}_{\mathcal{V}}.$$

The meaning of modalities is given by transitions. The formula $\langle a \rangle \alpha$ holds in some state if it has an outgoing a-transition to some state satisfying α. Dually, the formula $[a]\alpha$ holds in some state if all its outgoing a-transitions go to states satisfying α:

$$\left[\!\left[\langle a \rangle \alpha\right]\!\right]^{\mathcal{M}}_{\mathcal{V}} = \left\{ s \in S : \exists s'.R_a(s, s') \wedge s' \in [\![\alpha]\!]^{\mathcal{M}}_{\mathcal{V}} \right\},$$

$$\left[\!\left[[a]\alpha\right]\!\right]^{\mathcal{M}}_{\mathcal{V}} = \left\{ s \in S : \forall s'.R_a(s, s') \Rightarrow s' \in [\![\alpha]\!]^{\mathcal{M}}_{\mathcal{V}} \right\}.$$

Finally, the μ and ν constructs are interpreted as fixpoints of operators on sets of formulas. A formula $\alpha(X)$ containing a free variable X can be seen as an operator on sets of states mapping a set S' to the semantics of α when X is interpreted as S', in symbols: $S' \mapsto [\![\alpha]\!]^{\mathcal{M}}_{\mathcal{V}[S'/X]}$. Since by definition of the basic operators of the logic this operator is monotonic, it has well-defined least and greatest fixpoints. Formally,

$$[\![\mu X.\alpha]\!]^{\mathcal{M}}_{\mathcal{V}} = \bigcap \{ S' \subseteq S : [\![\alpha]\!]^{\mathcal{M}}_{\mathcal{V}[S'/X]} \subseteq S' \},$$

$$[\![\nu X.\alpha]\!]^{\mathcal{M}}_{\mathcal{V}} = \bigcup \{ S' \subseteq S : S' \subseteq [\![\alpha]\!]^{\mathcal{M}}_{\mathcal{V}[S'/X]} \}.$$

We will often write $\mathcal{M}, s, \mathcal{V} \models \alpha$ instead of $s \in [\![\alpha]\!]^{\mathcal{M}}_{\mathcal{V}}$. Moreover we will omit $\mathcal{V}$ or $\mathcal{M}$ if it is not important, or clear from the context.

Examples: The simplest formulas are just those of modal logic: $\langle a \rangle tt$ means "there is transition labeled by a". With one fixpoint, we can talk about termination properties of paths in a transition system. The formula $\mu X.[a]X$ means that all sequences of a-transitions are finite. The formula $\nu Y.\langle a \rangle Y$ means that there is an infinite sequence of a-transitions. We can then add a predicate p, and obtain $\nu Y.\, p \wedge \langle a \rangle Y$ saying that there is an infinite sequence of a-transitions, and all states in this sequence satisfy p. The formula $\mu X.\langle a \rangle X$ is just false, but the formula $\mu X.\, p \vee \langle a \rangle X$ says that there is a sequence of a-transitions leading to a state where p holds. With two fixpoints, we can write fairness formulas, such as $\nu Y.\, \mu X.\, (p \wedge \langle a \rangle Y) \vee \langle a \rangle X$ meaning "on some a-path there are infinitely many states where p holds". Changing the order of fixpoints we get $\mu X.\, \nu Y.\, (p \wedge \langle a \rangle Y) \vee \langle a \rangle X$ saying "on some a-path almost always p holds". To see why these formulas mean what they do, one can of course use the semantics directly, but it is often easier to use some alternative approaches that we introduce in the following. As these examples suggest, the semantics depends on the order of fixpoint operators, and the expressive power increases with the number of fixpoints (see Sect. 26.2.2).

Some Syntactic Conventions. The fixpoint operators μX and νX bind occurrences of the variable X, in the sense that the meaning of $\mu X.\alpha$ does not depend on the valuation of X. We leave to the reader the formal definition of bound and free occurrence of a variable in a formula. A *sentence* is a formula without free variables. In particular, the meaning of a sentence does not depend on the valuation of variables. By $\alpha[\beta/X]$ we denote the result of substitution of β for every free occurrence of X in α; when doing this we suppose that the free variables of β are disjoint from the bound variables of α. Clearly $\mu X.\alpha$ is equivalent to $\mu Y.(\alpha[Y/X])$, so we can always make sure that no variable has at the same time a free and a bound occurrence in a formula.

In order to underline the dependency of the value of α on X, we will often write $\mu X.\alpha(X)$ instead of $\mu X.\alpha$. In this context we write $\alpha(\beta)$ for $\alpha[\beta/X]$. We immediately employ this notation to introduce the idea of unfolding. A fixpoint formula $\mu X.\alpha(X)$ is equivalent to its *unfolding*, $\alpha(\mu X.\alpha(X))$. This is a very useful rule that allows us to "delay" reasoning about fixpoints. The equivalence of a formula with

its unfolding follows directly from the fact that μ is a fixpoint operator. Of course the same applies for ν in place of μ.

Semantics Based on Approximations. There is another very convenient way of defining meanings of fixpoint constructs. It comes directly from the Knaster–Tarski theorem characterizing fixpoints in a complete lattice in terms of their approximations. Let us start with a definition of formal approximations of fixpoint formulas: $\mu^\tau X.\alpha(X)$ and $\nu^\tau X.\alpha(X)$ for every ordinal τ. The meaning of $\mu^0 X.\alpha(X)$ is the empty set. The meaning of $\mu^{\tau+1} X.\alpha(X)$ is that of $\alpha(Z)$ where Z is interpreted as $\mu^\tau X.\alpha(X)$. Finally, the meaning of $\mu^\tau X.\alpha(X)$ when τ is a limit ordinal is the least upper bound of meanings of $\mu^\rho X.\alpha(X)$ for $\rho < \tau$. Similarly for $\nu^\tau X.\alpha(X)$ but for the fact that $\nu^0 X.\alpha(X)$ is the set of all states, and the greatest lower bound is taken when τ is a limit ordinal.

Example: Let us look at the meaning of approximations of a formula $\mu X.[a]X$:

$$\begin{aligned}
&\mu^0 X.[a]X = \varnothing && \text{false}\\
&\mu^1 X.[a]X = [a]\varnothing && \text{states with no } a\text{-path}\\
&\mu^2 X.[a]X = [a][a]\varnothing && \text{states with no } aa\text{-path}\\
&\qquad \cdots\\
&\mu^\omega X.[a]X = \bigcup_{n<\omega} \mu^n && \text{states s.t. } \exists n.\ \text{no } a^n\text{-path}\\
&\qquad \cdots
\end{aligned}$$

If every state has only finitely many a-successors, then the approximation *closes at* ω, i.e., $\mu^{\omega+1} = \mu^\omega$; but for infinite-branching systems we may need to go further, and the approximation closes at the least upper bound of ordinal heights of an a-tree in the system (Fig. 5). In general, the least ordinal such that $\mu^\tau X.\alpha = \mu^{\tau+1} X.\alpha$ in a transition system $\mathcal{M}$ is called the closure ordinal of $\mu X.\alpha$ in $\mathcal{M}$. The closure ordinal always exists; its cardinal is bounded by the cardinality of the transition system.

For a more complex example, consider the formula $\nu Y.\,\mu X.\,\langle a\rangle((p \wedge Y) \vee X)$ which is another way of writing the previously seen formula "along some a-path there are infinitely many states where p holds". Here we have to calculate the approximations of the ν formula, and during each such calculation, we have to calculate the approximations of the μ formula, *relative to the current* ν *approximation*. For ease of tabulation, write ν^τ for $\nu^\tau Y.\,\mu X.\,\langle a\rangle((p \wedge Y) \vee X)$, and $\mu^{\tau,\tau'}$ for $\mu^{\tau'} X.(\langle a\rangle((p \wedge Y) \vee X))[\nu^\tau / Y]$. Now we have, with some abuse of notation:

$$\begin{aligned}
&\nu^0 && S\\
&\quad \mu^{0,0} && \varnothing\\
&\quad \mu^{0,1} && \big[\!\big[\langle a\rangle\big((p \wedge S) \vee \mu^{0,0}\big)\big]\!\big] = \big[\!\big[\langle a\rangle p\big]\!\big]
\end{aligned}$$

$$
\begin{array}{ll}
\mu^{0,2} & \left[\!\left[\langle a\rangle\big((p \wedge S) \vee \mu^{0,1}\big)\right]\!\right] = \left[\!\left[\langle a\rangle\big(p \vee \langle a\rangle p\big)\right]\!\right] \\
\cdots & \\
\nu^1 = \mu^{0,\infty} & \langle a\rangle \text{eventually}(p) \\
\mu^{1,1} & \left[\!\left[\langle a\rangle\big((p \wedge \nu^1) \vee \varnothing\big)\right]\!\right] = \left[\!\left[\langle a\rangle\big(p \wedge \nu^1\big)\right]\!\right] \\
\mu^{1,2} & \left[\!\left[\langle a\rangle\big((p \wedge \nu^1) \vee \mu^{1,1}\big)\right]\!\right] = \left[\!\left[\langle a\rangle\big((p \wedge \nu^1) \vee \langle a\rangle\big(p \wedge \nu^1\big)\big)\right]\!\right] \\
\cdots & \cdots \\
\nu^2 = \mu^{1,\infty} & \text{eventually}\big(p \wedge \langle a\rangle \text{eventually}(p)\big) \\
\cdots & \cdots \\
\nu^\infty & \text{infinitely often } p
\end{array}
$$

In this example, "eventually p" means "on some a-path, p will occur". If the modality $\langle a\rangle$ were replaced by the $[a]$ modality, then it would mean "on every a-path, p will occur".

Negation. Since the syntax we propose does not have the negation operation, it is useful to see that negation can be defined in the language. We first define by induction on the structure a formal negation operation $\neg\alpha$ on formulas and then state that it has the required properties.

$$
\begin{array}{cc}
\neg(\neg p) = p & \neg(\neg X) = X \\
\neg(\alpha \vee \beta) = \neg\alpha \wedge \neg\beta & \neg(\alpha \wedge \beta) = \neg\alpha \vee \neg\beta \\
\neg\langle a\rangle\alpha = [a]\neg\alpha & \neg[a]\alpha = \langle a\rangle\neg\alpha \\
\neg\mu X.\alpha(X) = \nu X.\neg\alpha(\neg X) & \neg\nu X.\alpha(X) = \mu X.\neg\alpha(\neg X).
\end{array}
$$

Observe that when applying this translation to a formula without free variables, the final result has all variables occurring un-negated, because of the two negations introduced when negating fixpoint expressions.

Fact 1 (Definability of Negation) *For every sentence α, every transition system $\mathcal{M}$ over the set of states S, and every valuation $\mathcal{V}$*:

$$\left[\!\left[\neg\alpha\right]\!\right]^{\mathcal{M}}_{\mathcal{V}} = S - \left[\!\left[\alpha\right]\!\right]^{\mathcal{M}}_{\mathcal{V}}.$$

Examples: The negation of the "everywhere always p" formula $\nu X.\, p \wedge [a]X$ is $\neg\nu X.\, p \wedge [a]X = \mu X.\neg(p \wedge [a](\neg X)) = \mu X.\, \neg p \vee \neg[a](\neg X) = \mu X.\, \neg p \vee \langle a\rangle X$, the "eventually somewhere $\neg p$" formula.

For a more complicated example let us come back to $\nu Y.\, \mu X.\, (p \wedge \langle a\rangle Y) \vee \langle a\rangle X$ meaning "along some a-path there are infinitely many states where p holds". Its negation is $\mu Y.\, \nu X.\, (\neg p \vee [a]Y) \wedge [a]X$ expressing, in a slightly cryptic way, "on every path almost always $\neg p$".

Special Forms of Formulas. Let us mention some useful special forms of formulas. First, as we have noted above, we can require that bound and free variables are different. We can also require that every variable is bound at most once in a formula. If both of these are the case, we say the formula is *well-named*. Moreover, we can even ensure that in every formula $\mu X.\alpha(X)$ variable X appears only once in $\alpha(X)$. This is because $\mu X.\mu Y.\alpha(X, Y)$ is equivalent to $\mu X.\alpha(X, X)$. Similarly for $\nu X.\alpha(X)$.

Another useful syntactic property is *guardedness*. A variable Y is guarded in $\beta(Y)$ if all occurrences of Y are preceded (not necessary directly) by a modality. For example, Y is guarded in $\langle a\rangle\mu X.(X \wedge Y \wedge p)$. A formula is *guarded* if for every subformula $\sigma Y.\beta(Y)$, variable Y is guarded in $\beta(Y)$. It turns out that every formula is equivalent to a guarded formula.

The algorithm for constructing an equivalent guarded formula uses an operation of removing open occurrences of a variable. An occurrence of a variable is *open* if it is neither guarded, nor preceded by a fixpoint operator. To see how it works, consider a formula $\mu Y.\beta(Y)$ and suppose we want to obtain an equivalent formula without open occurrences of Y in $\beta(Y)$. For this it suffices to replace every open occurrence of Y in $\beta(Y)$ by ff. To see why this may work, observe that $\mu Y.\, Y \wedge \gamma(Y)$ is equivalent to ff while $\mu Y.\, Y \vee \gamma(Y)$ is equivalent to $\mu Y.\gamma(Y)$. Now, due to the laws of propositional logic, the formula $\beta(Y)$ is equivalent to $Y \wedge \gamma(Y)$ or $Y \vee \gamma(Y)$ for some $\gamma(Y)$ with no open occurrence of Y. We get that $\mu Y.\beta(Y)$ is equivalent to $\mu Y.Y \wedge \gamma(Y)$ or $\mu Y.Y \vee \gamma(Y)$. By the observation above, these in turn are equivalent to ff or to $\mu Y.\gamma(Y)$, respectively. The translation for $\nu Y.\beta(Y)$ is dual: open occurrences of Y are replaced by tt.

To convert a formula to a guarded formula we repeatedly remove open occurrences of variables, starting from innermost fixpoint formulas. For example, consider a formula $\nu Y.\mu X.\alpha(X, Y)$, where $\alpha(X, Y)$ does not have fixpoint subformulas. We first remove open occurrences of X in $\mu X.\alpha(X, Y)$. In the obtained formula, $\mu X.\alpha'(X, Y)$, all occurrences of X are guarded as the formula does not have proper fixpoint subformulas. In consequence, every occurrence of Y in $\alpha'(\mu X.\alpha(X, Y), Y)$ is either open or guarded. Hence we remove open occurrences of Y in $\nu Y.\alpha'(\mu X.\alpha(X, Y), Y)$ and obtain a guarded formula.

Fact 2 (Special Form of Formulas) *Every formula can be transformed into an equivalent guarded, well-named formula. Moreover, one can require that in every subformula of the form $\sigma X.\beta(X)$ variable X appears at most once in $\beta(X)$.*

As observed in [53], contrary to some claims in the literature, the transformation into a guarded form described above can induce an exponential growth in the size of the formula. It is not known whether there is a better transformation. Often it is enough to remove open occurrences of bound variables though, and this transformation does not increase the size of the formula. A way of avoiding exponential blowup is to use vectorial syntax described below [108].

Vectorial Syntax. The original syntax of the μ-calculus allows us to freely mix all types of operators. In some contexts it is more interesting to have a formula in a

prenex form where all the fixpoint operators are on the outside. This is possible in the vectorial syntax we will now introduce. Another advantage of this syntax is that it is in general more compact, as it allows the sharing of common subformulas.

A *modal formula* is a formula of the μ-calculus without fixpoint operators. As we do not allow negation of a variable in the syntax, this formula is positive. A sequence $\boldsymbol{\alpha} = (\alpha^1, \ldots, \alpha^n)$ of n modal formulas is a *vectorial μ-calculus formula of height n*. If $\mathbf{X} = X^1, \ldots, X^n$ is a sequence of n variables and $\boldsymbol{\alpha}$ a vectorial formula of height n then $\mu\mathbf{X}.\boldsymbol{\alpha}$ and $\nu\mathbf{X}.\boldsymbol{\alpha}$ are vectorial formulas of height n.

The meaning of a vectorial formula of height n is an n-tuple of sets of states. Apart from that, the semantics is analogous to the scalar (i.e., ordinary) μ-calculus. More precisely, if $\boldsymbol{\alpha} = (\alpha^1, \ldots, \alpha^n)$ is a sequence of modal formulas then its meaning in a model $\mathcal{M}$ with a valuation $\mathcal{V}$ is $[\![\alpha^1]\!]^{\mathcal{M}}_{\mathcal{V}} \times \cdots \times [\![\alpha^n]\!]^{\mathcal{M}}_{\mathcal{V}}$. Observe that with the variables $\mathbf{X}$ distinguished, the meaning of $\boldsymbol{\alpha}$ is a function from $\mathcal{P}(S)^n$ to $\mathcal{P}(S)^n$. The meaning of $\mu\mathbf{X}.\boldsymbol{\alpha}$ is then the least fixed-point of this function. Similarly for $\nu\mathbf{X}.\boldsymbol{\alpha}$.

It turns out that vectorial and scalar μ-calculi have the same expressive power. This is one more example of the remarkable closure properties of the logic. The translation from scalar to vectorial formulas is rather direct. One introduces a variable for every subformula and then writes a fixpoint formula in the obvious way. The obtained vectorial formula has the property that the first component of its meaning is exactly the meaning of the scalar formula. The translation in the other direction relies on repeated use of the so-called Bekič principle [10]:

$$\mu\begin{bmatrix} X \\ Y \end{bmatrix}.\begin{bmatrix} \alpha(X,Y) \\ \beta(X,Y) \end{bmatrix} = \begin{bmatrix} \mu X.\alpha\big(X, \mu Y.\beta(X,Y)\big) \\ \mu Y.\beta\big(\mu X.\alpha(X,Y), Y\big) \end{bmatrix}.$$

This principle allows us to eliminate the prefix of fixpoint operators. In the result we obtain a vector of formulas. The formula at the i-th coordinate will give the semantics of the i-th coordinate of the original vectorial formula.

Fact 3 (Vectorial Syntax) *Every μ-calculus formula can be converted into an equivalent vectorial formula. Every vectorial formula can be converted into an equivalent* (*scalar*) *μ-calculus formula.*

The translation from scalar to vectorial form does not yield a blowup in size. It is conjectured that vectorial formulas may be exponentially smaller than their scalar equivalents.

26.2.2 Alternation Depth

The examples above suggest that the power of the logic comes from fixpoint operators. While most useful properties can be expressed with few fixpoints, it is the

nesting of the two types of fixpoints that is the source of both expressive power and algorithmic difficulties. We introduce some notions to state this formally.

Let α be a well-named formula. So for every bound variable Y we have a unique subformula $\sigma Y.\beta_Y$ in α; and it makes sense to say that Y is a μ-variable or a ν-variable depending on the binder. This syntactic convention makes it easier to define the notion of alternation depth, otherwise we would need to refer to specific occurrences of variables.

Definition 1 (Alternation Depth) The *dependency order* on bound variables of α is the smallest partial order such that $X \leq_\alpha Y$ if X occurs free in $\sigma Y.\beta_Y$. The *alternation depth of a μ-variable* X in formula α is the maximal length of a chain $X_1 \leq_\alpha \cdots \leq_\alpha X_n$ where $X = X_1$, variables $X_1, X_3, \ldots$ are μ-variables and variables $X_2, X_4, \ldots$ are ν-variables. The alternation depth of a ν-variable is defined similarly. The *alternation depth of formula* α, denoted *adepth*(α), is the maximum of the alternation depths of the variables bound in α, or zero if there are no fixpoints.

Examples: The now-familiar "on some a-path there are infinitely many states where p holds" formula $\nu Y.\,\mu X.\,(p \wedge \langle a\rangle Y) \vee \langle a\rangle X$ is a canonical example of an alternation depth 2 formula since Y has alternation depth 2. Indeed $Y \geq X$ in the dependency order, and Y is a ν-variable while X is a μ-variable. In contrast, the "there is a path where p holds almost always" formula $\mu X.\,(\nu Y.\,(p \wedge \langle a\rangle Y)) \vee \langle a\rangle X$ has alternation depth 1, since X does not occur free in $\nu Y.\,(p \wedge \langle a\rangle Y)$ and in consequence has alternation depth 1.

The following fact gives a first good reason for this seemingly complicated definition.

Fact 4 (Alternation Depth and Unfolding) *A formula $\mu X.\beta(X)$ has the same alternation depth as its unfolding $\beta(\mu X.\beta(X))$. Similarly for the greatest fixpoint.*

Indeed, after renaming bound variables to avoid their repeated use, the dependency order of a sentence $\beta(\mu X.\beta(X))$ is the disjoint union of the dependency order for $\mu X.\beta(X)$ and that for $\beta(X)$. The number of alternations in the latter is not greater than in the former.

Alternation depth is a parameter that appears in many contexts. It is crucial in translations between the logic and automata. It induces a hierarchy with respect to expressive power. The complexity of all known model-checking algorithms depends exponentially on this parameter.

We will see alternation depth often in this chapter. At the moment let us only observe that this apparently technical definition becomes much more readable in vectorial syntax: the alternation depth is just the number of alternations between μ and ν in the prefix. It is tiresome but not difficult to check that the translations between scalar and vectorial formulas introduced in Sect. 26.2.1 preserve alternation depth.

There is a commonly seen alternative formulation, analogous to the definition of arithmetic hierarchy. Rather than talking about the "alternation depth", we may

classify formulas into a hierarchy of Σ_n^μ and Π_n^μ classes according to the nesting of fixpoint operators. So, for example, Σ_1^μ consists of formulas with only μ fixpoints, and Π_1^μ consists of formulas having only ν fixpoints. Then Σ_2^μ is the closure of Π_1^μ under Boolean operations, substitutions, and μ. Observe that unlike for arithmetic, we need to explicitly mention substitutions in the definition. Class Π_2^μ is defined similarly but using closure under ν. It can be shown that a formula has alternation depth n if and only if it is syntactically both in Σ_{n+1}^μ and in Π_{n+1}^μ.

Examples: The "always on every a-path p" formula $\nu X.p \wedge [a]X$ is a Π_1^μ formula, and the "on every a-path eventually p" formula $\mu X.p \vee [a]X$ is Σ_1^μ; both are alternation depth 1. However, the "on some a-path p holds almost always" formula $\mu X.(\nu Y.(p \wedge \langle a\rangle Y)) \vee \langle a\rangle X$ also has alternation depth 1 but it is neither Π_1^μ nor Σ_1^μ. It is Σ_2^μ because it can be obtained by substituting the (Π_1^μ and therefore) Σ_2^μ formula $\nu Y.(p \wedge \langle a\rangle Y)$ for Z in the (Σ_1^μ and therefore) Σ_2^μ formula $\mu X.Z \vee \langle a\rangle X$. It is also Π_2^μ, for the same reason. Note also that the alternation depth 2 formula "on some a-path there are infinitely many states where p holds" written $\nu Y.\mu X.(p \wedge \langle a\rangle Y) \vee \langle a\rangle X$ is Π_2^μ but *not* Σ_2^μ.

26.2.3 Semantics in Terms of Games

There are two good reasons why the μ-calculus has the right to its own chapter in this Handbook: expressive power and algorithmic properties. Indeed the logic can encode most of the other logics used in verification, and still algorithmically it is not substantially more difficult than the others. Nevertheless, the μ-calculus has been relatively slow in gaining acceptance, mainly because of its compact syntax. It is quite difficult to decode the meaning of a formula using the semantic clauses presented above. This is why the semantics in terms of games that we introduce here is conceptually very useful.

To see what we are aiming at, consider the formula $[a](p_1 \vee (p_2 \wedge p_3))$. Suppose that we want to verify that the formula holds in a state s of some transition system $\mathcal{M}$. We describe the verification process as a game between two players: Eve and Adam. The goal of Eve will be to show that the formula holds, while Adam aims at the opposite.

The game is presented in Fig. 1. The positions of Eve are pointed, and those of Adam are square. For example, the initial position belongs to Adam, and he has to choose there a state t reachable from s on an a-transition. The leaf position $t \models^? p_i$ is winning for Eve iff p_i holds in t. Looking at the game, it should be clear that the initial formula holds iff Eve has a strategy to reach a winning leaf. Her strategy is to choose in every position $t \models^? p_1 \vee (p_2 \wedge p_3)$ a disjunct that holds in t, if there is one.

To see a more challenging case consider the formula "infinitely often p on every path" $\nu Y.\mu Z.[a](Z \vee (p \wedge Y))$. Observe that apart from the two fixpoints the formula very much resembles the previous one. The game presented in Fig. 2 is also

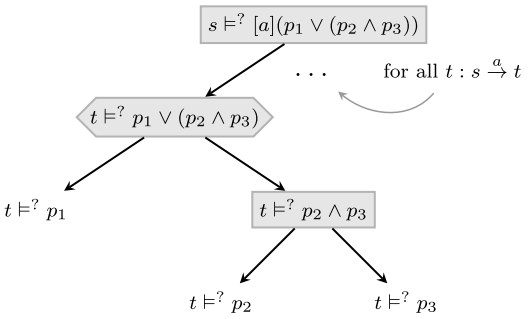


Fig. 1 Game for verifying $s \vDash^? [a](p_1 \vee (p_2 \wedge p_3))$

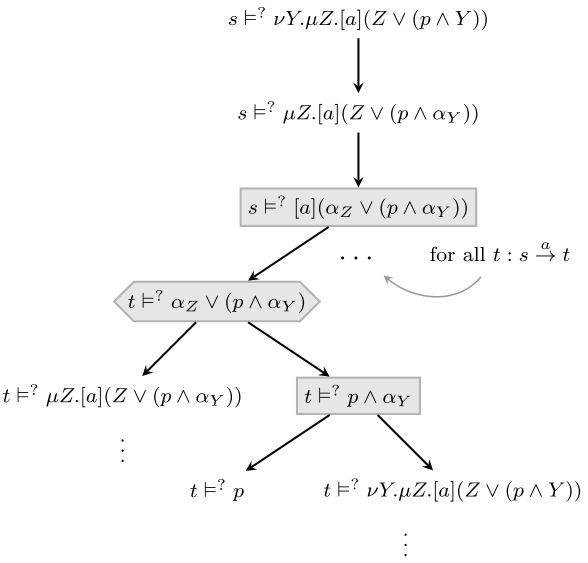
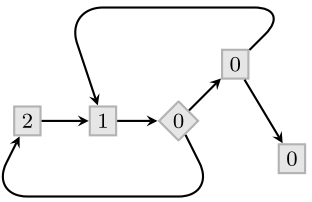


Fig. 2 Game for verifying $s \vDash^? \nu Y.\mu Z.[a](Z \vee (p \wedge Y))$

similar. For fixpoints we just apply the unfolding rule; we use α_Y to stand for the whole formula and α_Z for its subformula $\mu Z.[a](Z \vee (p \wedge Y))$. We have not marked to whom belong nodes with fixpoint formulas since it is not important: they always have a unique successor. Observe that this time the game may be infinite: from the bottom rightmost node we restart the whole game; from the bottom leftmost node we restart from the μ-subformula. The main point is to be able to decide who is the winner of an infinite play. As it will turn out, we cannot just say that all infinite plays are winning for one of the players. In this example, we will declare a path winning for Eve if the path passes infinitely often through the ν-formula (as in the rightmost bottom node).

In the following we will introduce the notion of a game with parity winning conditions, sufficient to deal with the μ-calculus. Parity conditions allow us to express properties like passing infinitely often through some node. Another chapter [19] of this Handbook describes many more variants of games used in model checking.

Fig. 3 A parity game



After presenting parity games, we will examine more closely the reduction of the model-checking problem to the problem of deciding the winner in a parity game constructed along the lines presented above.

26.2.3.1 Games

A *game* is a graph with a partition of nodes between two players, called Eve and Adam, and a set defining the winning condition. Formally it is a tuple

$$\mathcal{G} = \langle V, V_E, V_A, T \subseteq V \times V, Acc \subseteq V^\omega \rangle$$

where (V_E, V_A) is a partition of the set of nodes or *positions* V into those of Eve and those of Adam, T is the transition relation determining what are possible *successors* for each node, and Acc is a set defining the *winning condition*.

A *play* between Eve and Adam from some position $v \in V = V_E \cup V_A$ proceeds as follows: if $v \in V_E$ then Eve makes a choice of a successor, otherwise Adam chooses a successor; from this successor the same rule applies and the play goes on forever unless one of the parties cannot make a move. The player who cannot make a move loses. The result of an infinite play is an infinite path $v_0 v_1 v_2 \ldots$ This *path is winning* for Eve if it belongs to Acc. Otherwise Adam is the winner.

In the game presented in Fig. 3 the positions of Adam are marked with squares and the positions of Eve with diamonds. Additionally, each position is given a numerical *rank* in order to define Acc as we will see below. Observe that the unique position with no successors belongs to Adam, so he loses there. Let us say that Eve wins a play if it passes infinitely often through the position labeled with 2. For instance, if in the unique node for Eve she always chooses to go down, then she wins as 2 is on the loop. Actually Eve can also allow herself to go up, as then Adam has to go back to the position of rank 1. So as long as Eve goes down infinitely often she sees 2 infinitely often and wins.

A *strategy* for Eve is a function θ assigning to every sequence of nodes $\mathbf{v}$ ending in a node v from V_E a node $\theta(\mathbf{v})$ which is a successor of v. A *play respecting* θ is a sequence $v_0 v_1 \ldots$ such that $v_{i+1} = \theta(v_0 \ldots v_i)$ for all i with $v_i \in V_E$. The *strategy* θ *is winning for Eve* from a node v iff all the plays starting in v and respecting θ are winning. A *node is winning* if there exists a strategy winning from it. The strategies for Adam are defined similarly. A strategy is *positional* if it depends only on the last node in the sequence. So such a strategy can be represented as a function $\theta : V_E \to V$ and identified with a choice of edges in the graph of the game.

In this chapter we will consider only *parity winning conditions*. Such a condition is determined by a function $\Omega : V \to \{0, \ldots, d\}$ in the following way:

$$Acc = \left\{ v_0 v_1 \ldots \in V^\omega : \limsup_{i \to \infty} \Omega(v_i) \text{ is even} \right\}.$$

Hence, each position is assigned a natural number, called its *rank*, and we require that the largest rank appearing infinitely often is even. This condition, discovered by Mostowski [87] and independently by Emerson and Jutla [45], is the most useful condition in the context of the μ-calculus. The condition "infinitely often 2, or finitely often both 2 and 1" from the game in Fig. 3 is an example of a parity condition.

The main algorithmic question about such games is to decide who of the two players has a winning strategy from a given position. In other words to decide whether a given position is *winning for Eve* or *for Adam*. Principal results that we need about parity games are summarized in the following theorem. We refer the reader to [19, 114] for more details. We discuss complexity issues at the end of Sect. 26.2.4.

Theorem 5 (Solving Parity Games [45, 80, 88]) *Every position of a game with a parity winning condition is winning for one of the two players. Moreover, a player has a positional strategy winning from each of his winning positions. It is algorithmically decidable who is a winner from a given position in a finite game with a parity condition.*

26.2.3.2 Verification Game

We want to understand when a μ-calculus sentence α holds in a state s of a transition system $\mathcal{M}$. We characterize this by existence of a winning strategy in a specially constructed game $\mathcal{G}(\mathcal{M}, \alpha)$. More precisely, we want that $\mathcal{M}, s \vDash \alpha$ iff Eve has a winning strategy from a position corresponding to s and α in $\mathcal{G}(\mathcal{M}, \alpha)$. As we will need such a game for formulas with free variables as well, we will also take into account valuations. So, we will define a game $\mathcal{G}_{\mathcal{V}}(\mathcal{M}, \alpha)$, with $\mathcal{G}(\mathcal{M}, \alpha)$ being the special case when α is a sentence.

Positions in the game $\mathcal{G}_{\mathcal{V}}(\mathcal{M}, \alpha)$ are of the form (s, β) where s is a state of $\mathcal{M}$, and β is a formula from the *closure* of α, that is the smallest set containing α and closed under subformulas and unfolding. The intention is that Eve has a winning strategy from (s, β) iff $\mathcal{M}, s, \mathcal{V} \vDash \beta$.

We define the rules of the game by induction on the syntax of the formula (see Fig. 4). Clearly (s, p) should be declared winning for Eve if and only if proposition p holds in s. So we put no transition from this state and make it belong to Adam iff p holds in s. For a position $(s, \neg p)$ we proceed in the same way but exchange the roles of Adam and Eve. Observe that since there are no outgoing transitions, the player to whom the position belongs loses in it. For similar reasons a position

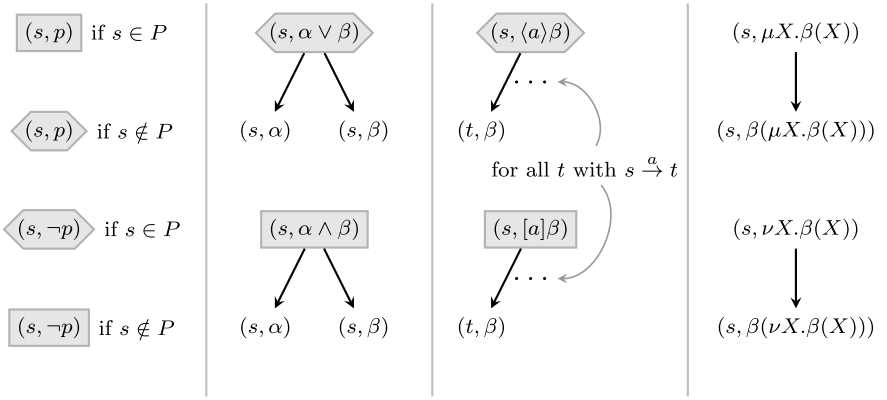


Fig. 4 Rules of the verification game $\mathcal{G}(\mathcal{M},\alpha)$

(s,X), with X a variable, has no outgoing transitions and it belongs to Adam iff $s \in \mathcal{V}(X)$.

From positions $(s,\alpha\vee\beta)$ and $(s,\alpha\wedge\beta)$ we put transitions to (s,α) and to (s,β). Position $(s,\alpha\vee\beta)$ should belong to Eve, as $\alpha\vee\beta$ is satisfied in a state if and only if at least one of α or β is satisfied. This means that Eve has to express her opinion on which of the two formulas holds. Dually, $(s,\alpha\wedge\beta)$ belongs to Adam.

From positions $(s,\langle a\rangle\beta)$ and $(s,[a]\beta)$ there are transitions to (t,β) for all t reachable from s by an a-transition, that is for all t such that $(s,t)\in R_a$. Position $(s,\langle a\rangle\beta)$ should belong to Eve as in order for a formula to be satisfied there should be an a-edge to some t satisfying β. Dually, $(s,[a]\beta)$ belongs to Adam.

Finally, from positions $(s,\mu X.\beta(X))$ and $(s,\nu X.\beta(X))$ there are transitions to $(s,\beta(\mu X.\beta(X)))$ and $(s,\beta(\nu X.\beta(X)))$ respectively. This corresponds to the intuition that a fixpoint is equivalent to its unfolding. As these positions have exactly one successor, it does not matter to which player they belong.

It remains to assign ranks to positions. We will be interested only in formulas starting with μ or ν; that is of the form $\mu X.\beta(X)$ or $\nu X.\beta(X)$. For a position with a formula of this form, we assign a rank in such a way that those starting with μ have odd ranks and those starting with ν have even ranks. Moreover, if γ is a subformula of β we require that the rank of γ is not bigger than that of β. One way to assign the ranks like this is to use the alternation depth (Definition 1).

$$\begin{aligned}
&\Omega(\gamma) = 2\cdot\lfloor adepth(X)/2\rfloor && \text{if } \gamma \text{ is of the form } \nu X.\gamma'(X)\\
&\Omega(\gamma) = 2\cdot\lfloor adepth(X)/2\rfloor + 1 && \text{if } \gamma \text{ is of the form } \mu X.\gamma'(X)\\
&\Omega(\gamma) = 0 && \text{otherwise}
\end{aligned}$$

Observe that we are using the alternation depth of X and not that of γ in this definition. This is because γ may contain formulas of big alternation depth not related to X. The sketch of the proof presented below provides more intuition behind this definition of rank.

Examples: Consider the example formulas from the end of Sect. 26.2.2. The rank of formula $\nu X.p \wedge [a]X$ is 0, while the rank of $\mu X.p \vee [a]X$ is 1. The rank of $\mu X.(\nu Y.(p \wedge \langle a\rangle Y)) \vee \langle a\rangle X$ is also 1 since the alternation depth of X is 1. But the rank of $\nu Y.\mu X.(p \wedge \langle a\rangle Y) \vee \langle a\rangle X$ is 2.

Having defined $\mathcal{G}_{\mathcal{V}}(\mathcal{M}, \alpha)$, and $\mathcal{G}(\mathcal{M}, \alpha)$ which is the special case when α is a sentence, we are ready to formulate one of the main theorems of this chapter.

Theorem 6 (Reducing Model Checking to Parity Games [45]) *For every sentence α, transition system $\mathcal{M}$, and its state s: $\mathcal{M}, s \vDash \alpha$ iff Eve has a winning strategy from the position (s, α) in $\mathcal{G}(\mathcal{M}, \alpha)$.*

The rest of this subsection is devoted to the proof of this theorem.

We fix $\mathcal{M}$ and α. For a valuation $\mathcal{V}$ we will denote by $\mathcal{G}_{\mathcal{V}}$ the game $\mathcal{G}_{\mathcal{V}}(\mathcal{M}, \alpha)$. By induction on the structure of the formula we show that for every state s and valuation $\mathcal{V}$:

$$\mathcal{M}, s, \mathcal{V} \vDash \beta \quad \text{iff} \quad \text{Eve wins from } (s, \beta) \text{ in } \mathcal{G}_{\mathcal{V}}.$$

The cases when β is a proposition or a variable follow directly from the definition. Among the other cases, the interesting ones are for the fixpoint operators. We will do the one for μ. In the proof we do not assume that $\mathcal{M}$ is finite, and this is why we need to consider ordinals.

Take a formula $\mu X.\beta(X)$ and consider left to right implication. Using the characterization of μ via approximations, we know that there is an ordinal τ such that $\mathcal{M}, s, \mathcal{V} \vDash \mu^{\tau} X.\beta(X)$. We can suppose that τ is the smallest such ordinal. Directly from definitions of approximations (see Sect. 26.2.1), it then follows that τ is a successor ordinal, so $\tau = \rho + 1$. In other words: $\mathcal{M}, s, \mathcal{V}_{\rho} \vDash \beta(X)$, where $\mathcal{V}_{\rho}$ is $\mathcal{V}$ modified so that the value of X is $[\![\mu^{\rho} X.\beta(X)]\!]^{\mathcal{M}}_{\mathcal{V}}$. We will do additional induction on ρ.

The outermost induction on the structure of the formula gives a winning strategy for Eve from $(s, \beta(X))$ in the game $\mathcal{G}_{\mathcal{V}_{\rho}}$. This strategy may reach a winning position (s', X) for some $s' \in \mathcal{V}_{\rho}(X)$. Since $\mathcal{V}_{\rho}(X) = [\![\mu^{\rho} X.\beta(X)]\!]^{\mathcal{M}}_{\mathcal{V}}$, the induction assumption on ρ tells us that Eve has a winning strategy from $(s', \mu X.\beta(X))$ in $\mathcal{G}_{\mathcal{V}}$. So the winning strategy for Eve in $\mathcal{G}_{\mathcal{V}}$ from $(s, \mu X.\beta(X))$ is to go to $(s, \beta(\mu X.\beta(X)))$ and then follow the winning strategy from $(s, \beta(X))$ in the game $\mathcal{G}_{\mathcal{V}_{\rho}}$. If a play respecting this strategy reaches $(s', \mu X.\beta(X))$, Eve can change to a winning strategy that exists there.

For the implication in the other direction, we suppose that Eve has a winning strategy from $(s, \mu X.\beta(X))$ in $\mathcal{G}_{\mathcal{V}}$. Note that this position has an odd rank, say r. The first crucial observation is a refinement of Fact 4. For every subformula $\sigma Y.\gamma(Y)$ of $\beta(\mu X.\beta(X))$ there are two possibilities: (i) either $\mu X.\beta(X)$ is not a subformula of $\gamma(Y)$, or (ii) $adepth(Y) \leq adepth(X)$ and the inequality is strict if Y is a ν-variable. Indeed suppose that $\mu X.\beta(X)$ is a subformula of $\sigma Y.\gamma(Y)$ that is itself a subformula of $\beta(\mu X.\beta(X))$. Then $\mu X.\beta(X)$ can be written as $\mu X.\theta(X, \sigma Y.\gamma'(X, Y))$, and the required inequalities between the alternation depths of X and Y follow by the definition. This observation would not be true if we

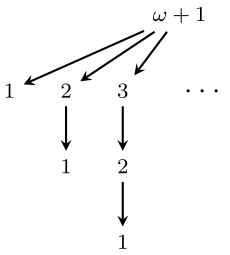


Fig. 5 A transition system where the closure ordinal of $\mu Z.[a]Z$ is infinite

had defined the rank of a position using the alternation depth of $\mu X.\beta(X)$ instead of the alternation depth of the variable X, for example, when $\mu X.\beta(X)$ has a subformula $\nu Y.\gamma(Y)$ that itself has a subformula with an alternation depth bigger than that of $\mu X.\beta(X)$.

The observation from the preceding paragraph implies that if a game from a position $(s, \mu X.\beta(X))$ reaches some position (s', β') with rank bigger than r then $\mu X.\beta(X)$ is not a subformula of β'. As the rank r is odd and every play is winning, this means that on every play there are finitely many positions of the form $(s', \mu X.\beta(X))$ for some s'. We call such positions *critical*. Consider a tree of all plays starting in $(s, \mu X.\beta(X))$ and respecting the winning strategy for Eve. We can assign to every critical position an ordinal, bounding the number of occurrences of critical positions on the paths starting from it. We call this ordinal the *height* of the position. All critical positions that do not have critical positions in their subtree will have height 1. Then, by induction, we take a critical position p such that all critical positions in its subtree already have a height assigned. Let τ be the least upper bound of these heights. We assign to p the height $\tau + 1$. It is not difficult to see that this procedure will assign a height to every critical position. Figure 5 gives an example of a tree where this procedure needs infinite ordinals.

Now, by induction on the ordinals assigned to critical positions we show that if $(s', \mu X.\beta(X))$ has height τ then $\mathcal{M}, s, \mathcal{V} \vDash \mu^{\tau}.\beta(X)$. First take a position $(s', \mu X.\beta(X))$ of height 1. This means that Eve has a winning strategy from this position that never enters another critical position. But then Eve also has a winning strategy from $(s', \beta(X))$ in the game $\mathcal{G}_{\mathcal{V}^1}$ where $\mathcal{V}^1$ is a valuation assigning $\varnothing$ to X. By induction hypothesis, $\mathcal{M}, s, \mathcal{V}^1 \vDash \beta(X)$, which is equivalent to $\mathcal{M}, s, \mathcal{V} \vDash \mu^1 X.\beta(X)$. By a similar argument if we take a critical position $(s', \mu X.\beta(X))$ of height $\tau + 1$ then Eve has a strategy winning from $(s', \beta(X))$ in the game $\mathcal{G}_{\mathcal{V}^\tau}$ where $\mathcal{V}^\tau$ is a valuation assigning to X the set of all nodes s'' such that $(s'', \mu X.\beta(X))$ has height at most τ. By induction hypothesis, $\mathcal{M}, s, \mathcal{V}^\tau \vDash \beta(X)$. From the induction on the height we can deduce that $\mathcal{V}^\tau(X) \subseteq [\![\mu^\tau X.\beta(X)]\!]^{\mathcal{M}}_{\mathcal{V}}$. We obtain $\mathcal{M}, s, \mathcal{V} \vDash \mu^{\tau+1} X.\beta(X)$. This completes the induction and the proof.

26.2.4 Model Checking

The model-checking problem for the μ-calculus is: given a sentence α, a transition system $\mathcal{M}$, and its state s, decide whether $\mathcal{M}, s \models \alpha$. Formulating the question in such a way, we silently assume that $\mathcal{M}$ is finite and given explicitly.

One approach to model checking is to directly use the μ-calculus semantics, or its variation through approximations. Since $\mathcal{M}$ is finite, there is a finite number of sets of states of $\mathcal{M}$ and such a computation will terminate.

The semantics via games gives a more efficient way to treat the problem. Observe that the size of the game $\mathcal{G}(\mathcal{M}, \alpha)$ is linear both in the size of $\mathcal{M}$ and in the size of α. So there is a linear-time reduction from the model-checking problem to deciding a winner in a parity game. We should remark that the reduction is also linear for formulas in vectorial syntax.

Theorem 7 (Equivalence of Games and Model Checking [42, 45]) *The model-checking problem is linear-time equivalent to the problem of deciding whether Eve has a winning strategy from a given position in a given parity game. The game constructed from a transition system of size m and a formula of size n has size $O(m \cdot n)$, the number of ranks in the game is equal to one more than the alternation depth of the formula. Conversely, from a game one can construct a transition system and a formula. The transition system is of the same size as the game. The formula depends only on the ranks used in the game, its size is linear in the number of ranks, and its alternation depth is not bigger than the number of ranks.*

The increase by one of the number of ranks in the game in the above theorem is less disturbing than it looks. Such an increase can appear for formulas that are both in Σ_n^μ and in Π_n^μ classes (see Sect. 26.2.2). For example, the formula $(\mu X.[a]X) \wedge (\nu Y.\langle a \rangle Y)$ is of alternation depth 1, and the constructed game needs rank 1 for μ, and rank 0 for ν (see Sect. 26.2.3.2). This does not really increase the complexity of solving the game as there will be no strongly connected component of the game graph containing both ranks 0 and 1. In general, in every connected component of a game constructed from a transition system and a formula of alternation depth d, the number of ranks in a connected component will be not bigger than d. Games with this property can be solved as quickly as games with d ranks.

The problem of solving parity games is in its turn equivalent to checking emptiness of alternating automata on infinite sequences over a one-letter alphabet. The latter is the same as checking emptiness of nondeterministic tree automata. For definitions of these types of automata we refer the reader to [59, 113, 120] and to the automata chapter of this Handbook [76]. Here we just state the corollary that is a consequence of these equivalences.

Theorem 8 (Equivalence of Model Checking and Automata Emptiness) *The model-checking problem is linear-time equivalent to checking emptiness of alternating parity word automata over a one-letter alphabet. It is also equivalent to checking emptiness of nondeterministic parity tree automata. The automata in question have*

the same number of states as there are nodes in the graph of the game from Theorem 7, and the same parity condition as the game.

Going back to Theorem 7, we briefly discuss the reduction from games to model checking as the other reduction is already provided by Theorem 6. It turns out that once the ranks are fixed, one can write a formula defining Eve's winning positions in a game with those ranks. Moreover, the size of this formula is linear in the number of ranks. More precisely, let us suppose that the ranks range from 0 to $2d+1$. A game $\mathcal{G} = \langle V, V_E, V_A, T \subseteq V \times V, \Omega : V \to \{0, \ldots, 2d+1\}\rangle$ can be represented as a transition system $\mathcal{M}_{\mathcal{G}}$ where V is the set of states, and T is a transition relation on some letter, say b. The additional information as to whether a position belongs to V_E and what is its rank is coded with auxiliary propositions: $p_{Eve}, p_{Adam}, p_0, \ldots, p_{2d+1}$. To write a formula defining the winning position we take variables $Z_0, \ldots, Z_{2d+1}$, one for every possible value of the rank. The formula is:

$$\mu Z_{2d+1}.\nu Z_{2d} \ldots \mu Z_1.\nu Z_0.\ \gamma(Z_0, \ldots, Z_d) \quad \text{where}$$

$$\gamma(Z_0, \ldots, Z_d) \text{ is } \bigwedge_{i=0,\ldots,2d+1} p_i \Rightarrow \big[\big(p_{Eve} \wedge \langle b\rangle Z_i\big) \vee \big(p_{Adam} \wedge [b] Z_i\big)\big]. \qquad (1)$$

The subformula $\gamma(Z_0, \ldots, Z_d)$ verifies the rank of a position using propositions p_i; this determines the fixpoint variable to be used. The formula also uses a $\langle b\rangle$ or $[b]$ modality depending on whether the position belongs to Eve or Adam, respectively. The alternation of fixpoints captures the parity condition. It can be verified that the above fixpoint formula defines the set of positions from which Eve has a winning strategy. Indeed the formula is constructed in such a way that the model-checking game $\mathcal{G}(\mathcal{M}_{\mathcal{G}}, \gamma)$ is essentially the same as $\mathcal{G}$. If the range of the Ω function is not as we have assumed, then it is enough to extend the range, write a formula for the extended range, and then remove the unnecessary fixpoint variables.

Hierarchical Boolean Equations. Games are not the only way to simplify the structure of the model-checking problem. It turns out that the problem is equivalent to checking whether a vectorial μ-calculus formula holds in a particularly simple model: the one containing just one state and no transitions. Such problems are known as *hierarchical Boolean equations* because the value of a variable is a Boolean: it can be either the empty set or the singleton containing the unique state.

Let $\mathcal{M}_0 = \langle \{s\}, \{R_a\}_{a\in Act}, \{P_i\}_{i\in\mathbb{N}}\rangle$ be the transition system with one state s and all transition relations and predicates empty: $R_a = \varnothing$, $P_i = \varnothing$. The meaning of every variable is then either $\varnothing$ or $\{s\}$. All formulas $\langle a\rangle\alpha$ are equivalent to ff, and formulas $[a]\alpha$ are equivalent to tt. Hence essentially we are left with Boolean connectives and fixpoints. Relying on the equivalence between games and model checking described above, the following fact states the announced reduction.

Fact 9 (Model Checking and Hierarchical Boolean Equations) *For every game $\mathcal{G}$ and its position v there is a vectorial formula $\alpha_{\mathcal{G},v}$, of height equal to the number of positions in $\mathcal{G}$ and size linear in the number of edges in $\mathcal{G}$, such that for the one-state*

transition system $\mathcal{M}_0$ described above: *the first coordinate of* $[\![\alpha_{\mathcal{G},v}]\!]^{\mathcal{M}_0}$ *is equal to* $\{s\}$ *iff* v *is winning for Eve in* $\mathcal{G}$.

We will briefly explain this reduction. Given a finite game

$$\mathcal{G} = \langle V, V_E, V_A, T \subseteq V \times V, \Omega : V \to \mathbb{N} \rangle$$

we write a vectorial formula of height $n = |V|$. We introduce variables X_i^v where $v \in V$ is a position, and $i \in \Omega(V)$ is one of the possible ranks. Then we define a formula:

$$\alpha^v = \begin{cases} \bigvee \{X^{v'}_{\Omega(v')} : T(v, v')\} & \text{if } v \text{ is a position of Adam,} \\ \bigwedge \{X^{v'}_{\Omega(v')} : T(v, v')\} & \text{if } v \text{ is a position of Eve.} \end{cases}$$

Let us fix some arbitrary order on positions V such that our chosen position is the first one in this order. We get a vectorial formula $\boldsymbol{\alpha} = (\alpha^{v_1}, \ldots, \alpha^{v_n})$. Let $\mathbf{X}_i$ stand for the vector $X_i^{v_1} \ldots X_i^{v_k}$. Then the desired formula is

$$\nu \mathbf{X}_d . \mu \mathbf{X}_{d-1} . \ldots \nu \mathbf{X}_0 . \boldsymbol{\alpha}. \tag{2}$$

Observe that this formula does not have free variables. While the presented translation is superficially quadratic, it should be clear that of the variables $X_1^v \ldots X_d^v$, only the one with the subscript $\Omega(v)$ is used. So the size of the formula not counting dummy variables is proportional to the number of edges in the game.

Complexity of Model Checking. The complexity of model checking is the great unanswered question about the modal μ-calculus. Effectiveness on finite systems is easy: the obvious algorithm based on the semantics via approximations has complexity $O(n^{d+1})$, where n is the size of the state space, and d is the alternation depth of the formula. The reduction to parity games quickly gives some new insights. To decide whether Eve wins from a given position it is enough to guess a strategy and check that it is winning. By Theorem 5 it is enough to consider only positional strategies, which are nothing else but subsets of edges of the game graph. It is not difficult see that one can check in a polynomial time whether a given positional strategy is winning. The algorithm essentially involves analysis of strongly connected components in the game graph. Consequently, solving the game is in NP, and since everything is closed under negation, also in co-NP. The obvious lower bound is PTIME since alternating reachability is a very simple parity game.

Most of the effort on complexity analysis of the model-checking problem has concentrated on the game formulation [17, 19, 42, 45, 47, 71, 72, 83, 102, 122, 125], although it is worth mentioning also the approaches through vectorial syntax and Boolean equations [78, 107]. In the discussion below let n stand for the number of vertices in the game and d for the number of ranks. In terms of the model-checking problem, n is the product of the sizes of the transition system and the formula, while d is the alternation depth of the formula.

One of the most classical approaches to solve parity games is based on removing positions of the highest rank and recursively analysing the resulting subgame [83, 113, 125]. Since the subgame is analysed twice, this approach gives $O(n^d)$ complexity. A different technique proposed by Jurdziński [71] is based on so-called progress measures and gives an $O(n^{\lceil d/2 \rceil})$ algorithm. More recently, Schewe [102] used a combination of the two to obtain an $O(n^{\lceil d/3 \rceil})$ algorithm. Better approaches are known if there are many priorities with respect to the number of nodes, more precisely if $d = \Omega(n^{(1/2)+\varepsilon})$. The randomized algorithm of Björklund et al. [17] gives $n^{O(\sqrt{n/\log(n)})}$ complexity. If the out-degree of all vertices is bounded then the same complexity is achieved by a relatively simple, and very clever, modification of McNaughton's algorithm [72] proposed by Jurdziński, Paterson, and Zwick. In the case when there is no bound on the degree, the same algorithm is only slightly worse: $n^{O(\sqrt{n})}$. For all of those algorithms superpolynomial or exponential lower bounds are known [52, 71, 102].

There exists another class of algorithms, based on strategy improvement techniques [65]. The idea is that one puts an order on positional strategies, so that the biggest elements in this order are the optimal strategies. The algorithm is an iteration of improvement steps: in each step some edges of the strategy are changed to improve with respect to the order on strategies. This technique has been used in many contexts. It has been adapted to parity games by Vöge and Jurdziński [122]. Later Schewe [103] has proposed a modification of the strategy improvement policy. Even for this improvement, Friedmann gives examples of games requiring exponentially many iterations [51].

It is actually not that surprising that the quest for polynomial-time algorithm for model checking is still on. The problem is closely related to other stubborn questions of a similar type, such as: solving mean pay-off games, discounted pay-off games, and turn based simple stochastic games [35, 70]. Not much more is known about fragments of the logic. The reduction to games gives an easy model-checking algorithm for the alternation depth 2 fragment. This algorithm is quadratic in the size of the formula and the model. No essentially better algorithms are known, but it is worth mentioning in this context a recent advance on related problems [31]. Let us note that model checking for alternation-free μ-calculus (alternation depth 1) can be done in linear time [3, 7, 33].

26.3 Fundamental Properties

In this section we will give an overview of the theory of the μ-calculus. Our intention is to cover a representative selection of results having in mind applications in verification. We will start with some basic results on theory of models: the tree-model property and bisimulation invariance. Then we will introduce modal automata: an automata model for the μ-calculus. We will discuss some basic features of this model, and in particular disjunctive form for automata. Disjunctive modal automata can be seen as a nondeterministic counterpart of modal automata. Among

other things, these automata allow us to treat the satisfiability problem. We present a relatively short proof of EXPTIME-completeness of the problem. On the way we also get a small-model theorem. Next, we briefly describe another central result of the theory: strictness of the alternation hierarchy. This is followed by the completeness theorem and two other properties of a more syntactical nature: the interpolation theorem and the division property.

26.3.1 Bisimulation Invariance and the Tree-Model Property

Bisimulation was introduced in the context of modal logics as an attempt to formulate the notion of similarity between models. Later it turned out that it is perfectly adapted to express the intuition that two systems behave in the same way. Formally, a *bisimulation* between two transition systems

$$\mathcal{M}^1 = \langle S^1, \{R_a^1\}_{a\in Act}, \{P_i^1\}_{i\in\mathbb{N}}\rangle \quad \text{and} \quad \mathcal{M}^2 = \langle S^2, \{R_a^2\}_{a\in Act}, \{P_i^2\}_{i\in\mathbb{N}}\rangle$$

is a relation $\approx$ between S^1 and S^2 such that if $s_1 \approx s_2$ then

- s_1 and s_2 satisfy the same propositions;
- for every action a, and s_1' such that $R_a^1(s_1, s_1')$ there is s_2' with $R_a^2(s_2, s_2')$ and $s_1' \approx s_2'$;
- and symmetrically, for every a, and s_2' such that $R_a^2(s_2, s_2')$ there is s_1' with $R_a^1(s_1, s_1')$ and $s_1' \approx s_2'$.

We say that a state of $\mathcal{M}^1$ is *bisimilar* to a state of $\mathcal{M}^2$ if there is a bisimulation relating the two states.

The intuition that the μ-calculus is about behaviors finds further support in the following result saying that the logic cannot distinguish between bisimilar states.

Theorem 10 (Invariance Under Bisimulation) *If a state s_1 of a transition system $\mathcal{M}^1$ is bisimilar to a state s_2 of a transition system $\mathcal{M}^2$ then for every μ-calculus sentence α: $\mathcal{M}^1, s_1 \vDash \alpha$ iff $\mathcal{M}^2, s_2 \vDash \alpha$.*

A consequence of this theorem is the tree-model property. A transition system is a (*directed*) *tree* if it has a state with a unique path to every other state in the transition system. This special state is called the *root*. A tree can be obtained as an *unfolding* of a transition system from a state s by taking all the paths starting in s as states of the unfolding. In the result, state s of the initial system and the root of the unfolding are bisimilar.

Proposition 1 (Tree Models) *Every satisfiable sentence of the μ-calculus is satisfied in a root of some tree transition system.*

Theorem 10 can be deduced from the semantics of the μ-calculus in terms of games. To see this, we define bisimulation relation between games as a bisimulation

between transition systems representing games (see Sect. 26.2.4). A parity game $\mathcal{G} = \langle V, V_E, V_A, T \subseteq V \times V, \Omega : V \to \mathbb{N} \rangle$ can be considered as a transition system with V being a set of states and T the transition relation on some action. The additional information as to whether a position belongs to V_E and the rank of a position are coded with auxiliary propositions: $p_{Eve}, p_{Adam}, p_0, \ldots, p_d$. So a bisimulation relation between games will be a bisimulation relation between their representations as transition systems. It is obvious that if a position v_1 in $\mathcal{G}^1$ is bisimilar to a position v_2 in $\mathcal{G}^2$ then Eve has a winning strategy from v_1 iff she has a winning strategy from v_2. The proof of Theorem 10 follows from the fact that if a state s_1 of $\mathcal{M}^1$ is bisimilar to a state s_2 of $\mathcal{M}^2$ then the positions (s_1, α) in $\mathcal{G}(\mathcal{M}^1, \alpha)$ and (s_2, α) in $\mathcal{G}(\mathcal{M}^2, \alpha)$ are bisimilar. This reasoning is a good example of the use of the semantics in terms of games. Let us point out though that Theorem 10 is one of the rare facts in the theory of the μ-calculus that can be proven directly by induction on the syntax of the formula, using approximations in the case of fixpoint formulas (see Sect. 26.2.1).

26.3.2 Modal Automata

The characterization of the semantics of the μ-calculus in terms of games suggests a kind of operational understanding of the logic. This idea is pushed a bit further here, by introducing an automata model that corresponds to the logic. As we will see the automata model is very close to formulas, but has its technical advantages. First, automata come with a notion of state and this helps in some constructions (Sects. 26.3.3 and 26.3.7). Moreover, modal automata have a very convenient special form called disjunctive modal automata. It is used, for example, to prove the interpolation property (Sect. 26.3.6). Modal automata can also be smoothly generalized to give extensions of the μ-calculus with good properties (Sect. 26.4.3).

Fix finite sets $\Sigma_A \subseteq Act$ of actions, $\Sigma_P \subseteq Prop$ for propositions, and Q of states. The set of *modal formulas* over these three sets, $\mathcal{F}_m(\Sigma_A, \Sigma_P, Q)$, is the smallest set containing $\Sigma_P \cup Q$ and closed under conjunction ($\alpha \wedge \beta$), disjunction ($\alpha \vee \beta$), and two modalities ($\langle b \rangle \alpha$ and $[b]\alpha$), for $b \in \Sigma_A$. Observe that modal formulas are just like μ-calculus formulas but without the fixpoint constructs.

A *modal automaton* is a tuple:

$$\mathcal{A} = \big\langle Q,\ \Sigma_A,\ \Sigma_P,\ q^0 \in Q,\ \delta : Q \to \mathcal{F}_m(\Sigma_A, \Sigma_P, Q),\ \Omega : Q \to \mathbb{N} \big\rangle.$$

It has a finite set of states Q, finite action and proposition alphabets Σ_A and Σ_P, one initial state q^0, and the parity acceptance condition given by Ω. The least standard part of the automaton is its transition function: the dependence on the alphabet is hidden in modal formulas.

Example: Let us write an automaton to represent the formula $\mu X.p_1 \vee \langle b \rangle X$. The alphabets of the automaton are $\Sigma_A = \{b\}$, $\Sigma_P = \{p_1\}$. It has one state q, and we let $\delta(q) = p_1 \vee \langle b \rangle q$. Since the formula uses the least fixed point we put $\Omega(q) = 1$.

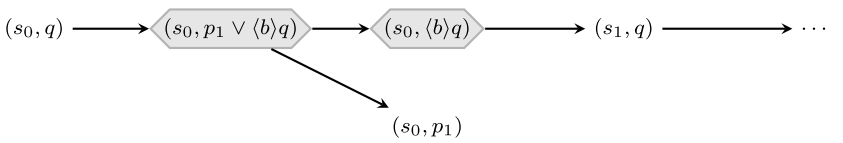


Fig. 6 The acceptance game $\mathcal{G}(\mathcal{M}, \mathcal{A})$ for a modal automaton

A modal automaton accepts transition systems with distinguished states. The acceptance is defined by games. Given a transition system $\mathcal{M} = \langle S, \{R\}_{a \in Act}, \{P_i\}_{i \in \mathbb{N}} \rangle$ we consider the *acceptance game* $\mathcal{G}(\mathcal{M}, \mathcal{A})$:

- The set of positions V of the game consists of pairs (s, α), where $s \in S$ is a state of $\mathcal{M}$ and α is a subformula of $\delta(q)$ for some $q \in Q$. A position (s, α) is for Eve if α is of one of the forms: q, $\beta' \vee \beta''$, or $\langle a \rangle \beta'$. Eve also has a position (s, p) if $s \not\models p$, and a position $(s, \neg p)$ if $s \models p$. The remaining positions are for Adam.
- From $(s, \alpha \vee \beta)$ and $(s, \alpha \wedge \beta)$ there are transitions to (s, α) and (s, β). From $(s, \langle a \rangle \alpha)$ and $(s, [a]\alpha)$ there are transitions to (t, α) for all t such that $(s, t) \in R_a$. From (s, q) there is a unique transition leading to $(s, \delta(q))$.
- The rank of a position of the form (s, q) is $\Omega(q)$, all other positions have rank 0.

The automaton *accepts* a transition system $\mathcal{M}$ from the state s iff Eve has a winning strategy in the game $\mathcal{G}(\mathcal{M}, \mathcal{A})$ from the position (s, q^0); recall that q^0 is the initial state of $\mathcal{A}$. We will denote this by $(\mathcal{M}, s) \in L(\mathcal{A})$, so the *language of* $\mathcal{A}$ is the set of transition systems with distinguished states.

Example: Let us take the automaton $\mathcal{A}$ from the previous example and some structure $\mathcal{M}$. Figure 6 shows the game $\mathcal{G}(\mathcal{M}, \mathcal{A})$. Starting in the position (s_0, q) the unique next position is $(s_0, p_1 \vee \langle b \rangle q)$. If p_1 holds in s_0 then Eve can choose this disjunct and win. Otherwise she chooses $\langle b \rangle q$, and then she chooses a successor s_1 of s_0 by a b-transition. The game reaches the position (s_1, q), and the whole reasoning repeats. It is possible that the game does not end. In this case Eve loses as the rank of q is odd. Observe that for this simple automaton Adam has no choice to make. Hence the automaton accepts iff there is a path of b-transitions leading to a state satisfying p_1.

Example: It may be instructive to see how nondeterministic automata on binary trees [113] can be represented as modal automata. A full labeled binary tree over an alphabet Σ_P is a function $t : \{l, r\}^* \to \Sigma_P$ with the empty word ε being the root and every word $w \in \{l, r\}^*$ being a node with a label $t(w)$, the left successor wl, and the right successor wr. Such a tree can be represented as a transition system $\mathcal{M} = \langle \{l, r\}^*, R_l, R_r, \{p\}_{p \in \Sigma_P} \rangle$. A transition function of a nondeterministic tree automaton has a shape $\delta_{\mathcal{B}} : Q \times \Sigma_P \to \mathcal{P}(Q \times Q)$, that is, to each state and letter it assigns a set of pairs of states: the state to send to the left child, and the state to send to the right child. The corresponding transition function for a modal automaton

becomes:

$$\delta_{\mathcal{A}}(q) = \bigvee_{p \in \Sigma_P} \left(p \wedge \bigvee_{(q_l, q_r) \in \delta(q,a)} \langle l \rangle q_l \wedge \langle r \rangle q_r \right).$$

The first element of the conjunct checks what is the label of the node, the second conjunct applies one of the possible transitions.

Equivalence with the μ-Calculus. We say that a modal automaton $\mathcal{A}$ is equivalent to a μ-calculus sentence α if for every transition system $\mathcal{M}$ and its state s:

$$(\mathcal{M}, s) \in L(\mathcal{A}) \quad \text{iff} \quad \mathcal{M}, s \vDash \alpha.$$

Thanks to the form of automata, it is easy to show that automata are equivalent to the logic.

Theorem 11 (Modal Automata [68]) *For every sentence of the μ-calculus there is an equivalent modal automaton. Conversely, for every modal automaton there is an equivalent sentence of the μ-calculus.*

It is worth looking at the constructions involved in this theorem. For the first statement of the theorem we take a formula α where every variable is bound at most once. This means that for every variable Y bound in α there is precisely one fixpoint subformula of α binding Y, which we denote by $\sigma Y.\beta_Y$.

We construct an equivalent automaton $\mathcal{A}_\alpha$. Let Σ_A be the set of actions appearing in α, let Σ_P be the set of propositions appearing in α. We take as Q the set containing q^0 and all bound variables appearing in α. The initial state of the automaton is q^0, and the acceptance condition Ω is defined using the notion of alternation depth (Definition 1). The value $\Omega(q_0)$ is irrelevant since the automaton will never come back to q_0; for the other states we have:

$$\Omega(Y) = 2 \cdot \lfloor adepth(Y)/2 \rfloor \qquad \text{if } Y \text{ is bound by } \nu \text{ in } \alpha,$$
$$\Omega(Y) = 2 \cdot \lfloor adepth(Y)/2 \rfloor + 1 \qquad \text{if } Y \text{ is bound by } \mu \text{ in } \alpha.$$

It remains to define the transition function of the automaton. For a subformula γ of α, we denote by $\widehat{\gamma}$ the formula obtained by replacing every fixpoint subformula $\sigma Y.\beta_Y$ by the variable Y. So $\widehat{\gamma}$ is a modal formula from $\mathcal{F}_m(\Sigma_A, \Sigma_P, Q)$. With the help of this notation we put:

$$\delta(q_0) = \widehat{\alpha}; \quad \text{and} \quad \delta(Y) = \widehat{\beta}_Y \text{ for every } Y \in Q.$$

In order to see that the automaton defined in such a way is equivalent to α, it suffices to observe that the evaluation game $\mathcal{G}(\mathcal{M}, \alpha)$ is isomorphic to the acceptance game $\mathcal{G}(\mathcal{M}, \mathcal{A})$; in the former valuation is irrelevant as α is a sentence.

For the second statement of Theorem 11 let us take an automaton $\mathcal{A} = \langle Q, \Sigma_A, \Sigma_P, q^0, \delta : Q \to \mathcal{F}_m(\Sigma_A, \Sigma_P, Q), \Omega : Q \to \mathbb{N} \rangle$. We construct the desired

formula in the vectorial syntax. The values of the transition function, $\delta(q)$, are from $\mathcal{F}_m(\Sigma_A, \Sigma_P, Q)$. These are formulas where the elements of Q play the role of variables. We introduce fresh variables X_i^q for every state $q \in Q$, and every possible rank $i \in \Omega(Q)$. Let α^q be the formula $\delta(q)$ with every state q' replaced by $X^{q'}_{\Omega(q')}$. With this notation we set $\boldsymbol{\alpha} = (\alpha^{q_0}, \ldots, \alpha^{q_n})$ where $q_0, \ldots, q_n$ is some enumeration of Q. The formula we are after is $\sigma \mathbf{X}_d \ldots \mu \mathbf{X}_1 . \nu \mathbf{X}_0 . \boldsymbol{\alpha}$; where the least fixed point is used to bind variables with odd index, and the greatest fixed point is used for variables with even index. Using the semantics of formulas in terms of games it is not difficult to show the equivalence of the formula with the automaton.

Closure Properties of Modal Automata. From the above theorem it follows that modal automata are closed under Boolean operations because the μ-calculus is. The proof also shows that one can directly use logical laws on transitions of modal automata without changing the accepted language. More precisely, if $\mathcal{A}_1$ and $\mathcal{A}_2$ are two automata over the same alphabet and the same set of states such that for every $q \in Q$, $\delta_1(q)$ is equivalent as a modal formula to $\delta_2(q)$, then $\mathcal{A}_1$ and $\mathcal{A}_2$ accept the same structures. This observation allows us to simplify transitions of automata.

As a side remark let us observe that unlike nondeterministic automata on binary trees, modal automata are not closed under projection. This is not a surprise since we want the automata to be equivalent to the μ-calculus. The automata, and logic, are closed under a weaker form of projection as explained in the interpolation section below.

Disjunctive Normal Form for Automata. Modal automata are essentially alternating automata [76, 89]. This is so because conjunction appearing in modal formulas permits the encoding of universal choice. In some contexts though, nondeterministic automata are much easier to handle than alternating automata. It is well known that over binary trees every alternating automaton can be converted to a nondeterministic one. Here we present a similar result for modal automata.

A simple solution to define nondeterministic automata would be to disallow conjunctions in modal formulas. This is not satisfactory though, as we would have no way to write properties like $(\langle a \rangle p_1) \wedge (\langle a \rangle p_2)$. There is an interesting modal operator that allows control over the use of conjunction. If Γ is a finite set of formulas and a is an action then

$$(a \to \Gamma) \quad \text{stands for} \quad \Big(\bigwedge \{\langle a \rangle \alpha : \alpha \in \Gamma\}\Big) \wedge [a]\Big(\bigvee \Gamma\Big).$$

We adopt the convention that the conjunction of the empty set of formulas is equivalent to tt, and its disjunction is equivalent to ff. So for example $(a \to \varnothing)$ says that there are no successors on action a. The new operator can express both existential and universal modalities:

$$\langle a \rangle \alpha \text{ is } \big(a \to \{\alpha, tt\}\big) \quad \text{and} \quad [a]\alpha \text{ is } \big(a \to \{\alpha\}\big) \vee (a \to \varnothing).$$

This operator was introduced in [68] and independently in [86] in the context of a coalgebraic approach to modal logic.

Define a *disjunctive formula* to be a disjunction of formulas of the form $\alpha \wedge \bigwedge_{a \in \Sigma}(a \to \Gamma_a)$ where α is a conjunction of propositions and each Γ_a is a set of states. Let $\mathcal{DF}(\Sigma_A, \Sigma_P, Q)$ denote the set of disjunctive formulas over the three alphabets. A *disjunctive modal automaton* is a modal automaton using disjunctive modal formulas in its transitions.

Theorem 12 (Disjunctive Modal Automata [68]) *Every modal automaton is equivalent to a disjunctive modal automaton.*

As we have noticed, modal automata may exhibit alternating behavior. Let us see why disjunctive modal automata have only nondeterministic behavior, that is they do not have universal branching. For this we need to look at their behavior on some transition system $\mathcal{M}$ that is a tree. The crucial property is that for every strategy of the automaton in the semantics game induced by $\mathcal{M}$, if two plays respecting the strategy differ at some position then they do not visit the same state of $\mathcal{M}$ after this position. So in the tree of all plays respecting a strategy, no state of $\mathcal{M}$ can appear in two different subtrees. This property corresponds to the fact that a run of a nondeterministic automaton on a binary tree can be presented as a labeling of the tree with states. Observe, incidentally, that the translation of nondeterministic tree automata to modal automata presented in the third example of this subsection in fact gives a disjunctive modal automaton. Similarly to (general) nondeterministic automata, the emptiness problem is easier for disjunctive modal automata: it suffices to guess a subgraph of the transition graph of the automaton. Complexity-wise the problem is in NP, while the emptiness problem for modal automata is EXPTIME-complete.

26.3.3 Satisfiability

Suppose that we want to decide whether given two formulas are equivalent. For this we should decide whether the two formulas are satisfied in the same set of models; in other words, decide whether the logical equivalence of the two formulas holds in every model. Thus formula equivalence is nothing else than the satisfiability problem: deciding whether there exists a model and a state where the formula is true. In this subsection we will discuss what is needed to solve the satisfiability problem.

Theorem 13 (Satisfiability [44, 46]) *The satisfiability problem for the μ-calculus is* EXPTIME*-complete.*

Using the correspondence between modal automata and μ-formulas (Theorem 11) instead of studying the satisfiability problem we can study the emptiness problem for modal automata: a formula is satisfiable iff the language of the modal automaton is not empty. Our goal is to reduce the emptiness problem for modal automata to the same problem for alternating automata on binary trees. Although

it would be much simpler to use disjunctive automata, we do not follow this path, as we have not discussed how to obtain disjunctive automata. Indeed, a translation to disjunctive automata would contain all the complexity of the satisfiability problem. The argument we present below is an example of an interesting technique of simplifying a problem by enriching transition systems with additional information.

Recall that a modal automaton is a tuple

$$\mathcal{A} = \langle Q, \Sigma_A, \Sigma_P, q^0 \in Q, \delta : Q \to \mathcal{F}_m(\Sigma_A, \Sigma_P, Q), \Omega : Q \to \mathbb{N} \rangle,$$

where $\mathcal{F}_m(\Sigma_A, \Sigma_P, Q)$ is a set of modal formulas over actions in Σ_A, propositions in Σ_P, and variables in Q. Acceptance is defined in terms of the existence of a winning strategy for Eve in the game $\mathcal{G}(\mathcal{M}, \mathcal{A})$; see Sect. 26.3.2.

Our first step will be to define witnesses for the existence of such a winning strategy. A witness will be a deterministic transition system over a bigger alphabet such that all its paths satisfy certain conditions (Lemma 1). Then we will encode such witnesses into a binary tree, and construct an alternating automaton recognizing all encodings of all possible witnesses. This will give a reduction of the satisfiability problem to the emptiness problem for alternating automata on binary trees.

The whole argument crucially depends on the fact that in parity games it is enough to consider positional strategies (Theorem 5). Positions in the acceptance game $\mathcal{G}(\mathcal{M}, \mathcal{A})$ are of the form (s, α) where s is a state of $\mathcal{M}$ and α a subformula of $\delta(q)$ for some $q \in Q$. In particular, the set of formulas appearing in positions of the game is finite. A positional strategy for Eve makes two kinds of choices: (i) in a position of the form $(s, \alpha \vee \beta)$ it chooses (s, α) or (s, β); (ii) in a position of the form $(s, \langle a \rangle \alpha)$ it chooses (s', α) for some state s'. So for a fixed state s the information provided by the strategy is finite: for every disjunction one disjunct is chosen, for every diamond formula $\langle a \rangle \alpha$ a successor state is chosen. These choices can be encoded in the label of a node, and we will still have only a finite number of labels.

Take a positional strategy σ for Eve in $\mathcal{G}(\mathcal{M}, \mathcal{A})$. We introduce new propositions and actions. A proposition $p^{\alpha}_{\alpha \vee \beta}$ will hold in s when $\sigma(s, \alpha \vee \beta) = (s, \alpha)$. A transition on action $b_{\langle a \rangle \alpha}$ from s to s' will mean that $\sigma(s, \langle a \rangle \alpha) = (s', \alpha)$. Let $\widehat{\Sigma}_P$ and $\widehat{\Sigma}_A$ be the alphabets of these new propositions and actions. Let $\sigma(\mathcal{M})$ stand for the transition system obtained from $\mathcal{M}$ by adding the new propositions and transitions in the way we have described.

We claim that looking at $\sigma(\mathcal{M})$ we can decide whether σ is winning in $\mathcal{G}(\mathcal{M}, \mathcal{A})$. To make this precise we define a notion of a *trace*. This is a sequence of pairs (s, α) consisting of a state of $\sigma(\mathcal{M})$ and a formula, such that:

- (s, q) is followed by $(s, \delta(q))$;
- $(s, \alpha \wedge \beta)$ is followed by (s, α) or (s, β);
- $(s, \alpha \vee \beta)$ is followed by (s, α) if $p^{\alpha}_{\alpha \vee \beta}$ holds in s, and by (s, β) otherwise;
- $(s, \langle a \rangle \alpha)$ is followed by (s', α) if there is transition from s to s' on action $b_{\langle a \rangle \alpha}$;
- $(s, [a]\beta)$ is followed by (s', β) if for some α there is transition from s to s' on action $b_{\langle a \rangle \alpha}$.

Examining this definition one can observe that a trace in $\sigma(\mathcal{M})$ is just a play in $\mathcal{G}(\mathcal{M}, \mathcal{A})$ respecting the strategy σ. So we can say that a trace is *winning* if it is

winning when considered as a play in $\mathcal{G}(\mathcal{M}, \mathcal{A})$. Finally, we say that $\sigma(\mathcal{M})$ is *trace-accepting from* (s^0, q^0) if all the traces starting in (s^0, q^0) are winning.

The first observation is that without loss of generality we can assume that $\sigma(\mathcal{M})$ is *deterministic*, in the sense that from every state, there is at most one outgoing transition on each action. Indeed, if $\sigma(\mathcal{M})$ is trace-accepting and it has a state with two outgoing transitions on action $b_{\langle a\rangle\alpha}$ then we can just remove one of them. This operation cannot add new traces, so the resulting structure is trace-accepting: the structure is even of the form $\sigma'(\mathcal{M})$ for some strategy σ'.

Lemma 1 *For our fixed modal automaton $\mathcal{A}$ and alphabet $\widehat{\Sigma}$, there is a transition system accepted by $\mathcal{A}$ iff there is a deterministic transition system over $\widehat{\Sigma}$ that is trace-accepting.*

We have defined $\sigma(\mathcal{M})$ and the notion of a trace in such a way that if σ is a positional winning strategy in $\mathcal{G}(\mathcal{M}, \mathcal{A})$ then $\sigma(\mathcal{M})$ is trace-accepting. As noted above, $\sigma(\mathcal{M})$ can be made deterministic. This shows left to right implication of the lemma.

For the converse implication, let us take a deterministic transition system $\mathcal{N}$ over the alphabet $\widehat{\Sigma}$. We define the structure $\mathcal{M}'$ by keeping the states of $\mathcal{N}$ and putting a transition on a from s to s' if there is a transition from s to s' on $b_{\langle a\rangle\beta}$ for some formula β. The positional strategy σ' in the game $\mathcal{G}(\mathcal{M}', \mathcal{A})$ can be read from $\mathcal{N}$ as follows: $\sigma'(s, \alpha \vee \beta) = \alpha$ iff s satisfies $p^{\alpha}_{\alpha\vee\beta}$ in $\mathcal{N}$, and $\sigma'(s, \langle a\rangle\alpha) = (s', \alpha)$ iff there is a transition from s to s' on $\langle a\rangle\alpha$. It can be verified that $\sigma(\mathcal{M}')$ is isomorphic to $\mathcal{N}$. Moreover every play in $\mathcal{G}(\mathcal{M}', \mathcal{A})$ respecting σ' and starting from a position (s, q) is a trace in $\mathcal{N}$ starting in (s, q). Hence if $\mathcal{N}$ is trace-accepting from (s^0, q^0) then σ' is winning from (s^0, q^0). Since q^0 is the initial state of $\mathcal{A}$, this means that the pair $(\mathcal{M}', s^0)$ is accepted by $\mathcal{A}$.

The above lemma permits us to reduce the satisfiability problem to the problem of finding a deterministic structure $\mathcal{N}$ over the alphabet $\widehat{\Sigma}$ that is trace-accepting. Observe that if $\mathcal{M}$ is a tree then $\sigma(\mathcal{M})$ is also a tree. Hence, by the tree-model property, Corollary 1, if there is such a structure $\mathcal{N}$ then there is one that is a deterministic tree. Since the set of labels of transitions is finite, $\mathcal{N}$ is a tree of bounded degree.

We have reduced the satisfiability problem to the problem of finding a tree of bounded degree satisfying the trace-acceptance condition. Using a straightforward encoding of bounded degree trees into binary trees we can reduce the problem to a question about full binary trees. So finally we need to construct an automaton recognizing binary trees that are encodings of bounded degree trees satisfying the trace-acceptance condition. Such an automaton $\mathcal{B}_3$ can be constructed in three steps: (i) taking a simple nondeterministic parity automaton $\mathcal{B}_1$ recognizing paths having a trace that is not winning; (ii) dualizing $\mathcal{B}_1$ to obtain an alternating parity automaton $\mathcal{B}_2$ recognizing paths on which all traces are winning; and (iii) constructing an alternating parity tree automaton $\mathcal{B}_3$ running $\mathcal{B}_2$ on every path of a tree.

Theorem 14 (Satifiability via Automata Emptiness [45]) *For a given μ-calculus formula of size n and alternation depth d, one can construct in linear time an alternating automaton over trees such that the formula is satisfiable iff there is a tree accepted by the automaton. The automaton has $O(n)$ states and $d+1$ ranks.*

This theorem gives an algorithmic solution to the satisfiability problem. As the emptiness check for such automata can be done in EXPTIME, this gives an upper bound on the complexity of the satisfiability problem, as announced in Theorem 13. The matching lower bound can be obtained, for example, by encoding of the universality problem for root-to-leaves automata over finite trees [106].

The above construction also proves the small-model theorem. A regular tree can be presented as a graph with a distinguished root vertex: the tree is obtained by *unfolding* such a graph, that is taking all the paths starting at the root. If an alternating automaton accepts some tree then it accepts a regular tree that is the unfolding of a transition system of size exponential in the size of the automaton.

Theorem 15 (Small-Model Property [112]) *A satisfiable formula of the μ-calculus is satisfied in a transition system of size exponential in the size of the formula.*

26.3.4 Alternation Hierarchy

We have seen that the alternation depth (Definition 1) of a formula appears to give a strong measure of its complexity, both psychological and computational: formulas rapidly become incomprehensible above alternation depth 2, and all algorithms known so far depend exponentially or super-polynomially on the alternation depth (Sect. 26.2.4). Recall the definition of sets Σ_n^μ and Π_n^μ from Sect. 26.2.2. Roughly, Σ_n^μ is the set of formulas of alternation depth n where all variables of alternation depth n are μ-variables. Similarly for Π_n^μ but for ν-variables.

For a number of years, it was open whether the alternation hierarchy is strict in terms of expressive power, that is, whether for every n there is a Σ_n^μ formula that is not equivalent to any Π_n^μ formula—and consequently, whether alternation depth $n+1$ is strictly more expressive than alternation depth n. The first proof by Bradfield [27] used the standard technique of diagonalization, via a transfer to arithmetic, relying on the small-model property for the transfer back. Subsequently, Arnold [6] gave a version of the proof in which diagonalization was effected with a topological argument—as this is probably the simplest proof, we will sketch it here.

Theorem 16 (Strictness of the Alternation Hierarchy [27]) *For every $n > 0$, there is a formula in Σ_n^μ which is not equivalent to any formula of Π_n^μ.*

The proof relies on the semantics of formulas in terms of games (Sect. 26.2.3). Recall that by Theorem 6 for a transition $\mathcal{M}$ and a formula β we can construct a game $\mathcal{G}(\mathcal{M}, \beta)$ such that $\mathcal{M}, s \vDash \beta$ if and only if Eve has a winning strategy from

the node (s, β) in $\mathcal{G}(\mathcal{M}, \beta)$. If $\beta \in \Sigma_n^\mu$ then $\mathcal{G}(\mathcal{M}, \beta)$ is a parity game over the ranks $\{1, \ldots, n\}$ or $\{0, \ldots, n-1\}$; depending on whether n is odd or even, respectively. Let us suppose n is odd, the argument for the other case is very similar. As we have discussed in Sect. 26.2.4, the game $\mathcal{G}(\mathcal{M}, \beta)$ can be represented as a transition system. A small but crucial observation is that if we start with a sufficiently large alphabet then we can actually suppose that this new transition system is over the same alphabet as $\mathcal{M}$. Hence a formula β determines a function F_β on transition systems over some fixed alphabet: for a transition system $\mathcal{M}$, the system $F_\beta(\mathcal{M})$ is the transition system representing the game $\mathcal{G}(\mathcal{M}, \beta)$.

We have supposed that games we consider are over ranks $\{1, \ldots, n\}$. Now recall the formula (1) from Sect. 26.2.4 defining the set of winning positions for Eve in a parity game over priorities $\{0, \ldots, n\}$. Let us call this formula α_n. This means that for every position (s, γ) in game $\mathcal{G}(\mathcal{M}, \beta)$, formula α_n holds in (s, γ) iff (s, γ) is winning for Eve. Combining this with the previous paragraph we get that for every transition system $\mathcal{M}$ and formula $\beta \in \Sigma_d^\mu$, $\mathcal{M}, s \vDash \beta \quad$ iff $\quad F(\mathcal{M}), (s, \beta) \vDash \alpha_n$. For our argument we will need a slightly more refined construction that works on tree transition systems: transition systems whose underlying graph is a tree. We fix a sufficiently big alphabet and a variation of formula α_n adapted to this alphabet, and construct a function F'_β such that for every tree transition system $\mathcal{N}$:

$$\mathcal{N}, \mathit{root} \vDash \beta \quad \text{iff} \quad F'_{\mathcal{B}}(\mathcal{N}), \mathit{root} \vDash \alpha'_n \ ;$$

here *root* stands for the root node of a tree transition system. We will omit the exact definition of F'_β. Formula α_n belongs to Σ_n^μ, and the same will be true for formula α'_n. We will show that it cannot belong to Π_n^μ.

Trees can be equipped with the usual metric: trees are 2^{-n} apart if they first differ at a node of depth n. The set of infinite trees with this metric is a complete metric space. So the mapping F'_β described above is a mapping on a complete metric space. It turns out that this mapping is contracting. The Banach Fixed Point Theorem says that every contracting mapping on a complete metric space has a fixpoint. In our case this means that for every formula β there is a tree transition system $\mathcal{N}_\beta$ such that $\mathcal{N}_\beta = F'_\beta(\mathcal{N}_\beta)$.

We want to show that $\alpha_n \notin \Pi_n^\mu$. Suppose conversely. Then there is a formula $\gamma \in \Sigma_n^\mu$ equivalent to $\neg\alpha_n$. We consider the mapping F'_γ and its fixpoint $\mathcal{N}_\gamma$. We get $\mathcal{N}_\gamma, \mathit{root} \vDash \gamma$ iff $F'_\gamma(\mathcal{N}_\gamma), \mathit{root} \vDash \alpha_n$. But $\mathcal{N}_\gamma$ is a fixpoint of F'_γ so $F'_\gamma(\mathcal{N}_\gamma) = \mathcal{N}_\gamma$; a contradiction.

This proof shows concrete examples of formulas at each level of the hierarchy: the formulas α_n expressing existence of a winning strategy. Nevertheless the hierarchy is far from being well understood. We do not know for example whether the semantic alternation depth, the smallest alternation depth of an equivalent formula, is a decidable property. This is an area of active research, which is mostly formulated in terms of automata theory rather than the μ-calculus.

26.3.5 Proof System

In order to show that a formula is satisfiable it is enough to exhibit a model for it. Theorem 15 tells us that every satisfiable formula has a finite model. So there is always a finite witness of satisfiability. The question arises, what about unsatisfiability? Since satisfiability is a decidable property (Theorem 13) the run of the algorithm is such a witness. A proof system gives a much more informative witness, moreover it gives reasoning principles that can be used to simplify formulas. To look at the question more positively, one prefers to talk about validity instead of unsatisfiability: a formula is *valid* if its negation is not satisfiable.

Validity of μ-calculus formulas can be axiomatized. This means that for a valid formula one can provide a finite witness, a proof, of its validity. To describe the proof system we will use inequalities $\alpha \leq \beta$. We will say that such an inequality is *valid* if the implication $\alpha \Rightarrow \beta$ is valid. In other words, whenever under some valuation formula α is true in some state of a model then β is true too in the same state with the same valuation. For example, a formula α is valid iff $tt \leq \alpha$ is valid; and a formula β is satisfiable iff $\beta \leq ff$ is not valid. We present a finitary proof system allowing us to deduce all valid inequalities. As examples of useful inequalities consider:

$$\mu X.\alpha \leq \nu X.\alpha \qquad \mu X.\nu Y.\alpha \leq \nu Y.\mu X.\alpha$$

The proof system proposed by Kozen [75] consists of axioms and rules for modal logic together with an axiom and a rule determining the semantics of the least fixpoint:

$$\alpha(\mu X.\alpha) \leq \mu X.\alpha$$

$$\frac{\alpha[\beta/X] \leq \beta}{\mu X.\alpha(X) \leq \beta}$$

The last rule expresses rather directly the semantic clause defining the least fixpoint (see Sect. 26.2.1). Additionally to these two, there is a dual axiom and a rule for the greatest fixpoint operator. This system is finitary as it contains only a finite number of axioms and rule schemes. The system is sound and complete in the sense that it proves exactly the valid inequalities.

Theorem 17 (Completeness [123]) *For every formula α, there is a proof of $tt \leq \alpha$ in the system if and only if the formula is valid.*

26.3.6 Interpolation Property

Craig interpolation is a desirable property of a logic. It testifies some kind of adequacy between the syntax of the logic and its semantics. On a more practical side,

it allows us to simplify formulas by just looking at their vocabulary [82]. Interpolation properties are scarce. Many program logics, such as LTL, CTL, and CTL*, for example, do not have the interpolation property [79]. Propositional logic and modal logic do have the interpolation property. So does the μ-calculus.

In order to simplify the presentation, let us consider here only the μ-calculus where the only propositions are *tt* and *ff*. In this case, the Craig interpolation property says that given two formulas α and β over sets of actions Σ_1 and Σ_2 respectively, if $\alpha \Rightarrow \beta$ is valid then there is a formula γ over the set of common actions $\Sigma_1 \cap \Sigma_2$ such that $\alpha \Rightarrow \gamma \Rightarrow \beta$. Actually an even stronger version of interpolation holds for the μ-calculus. In this version γ does not depend on β but only on its alphabet.

Theorem 18 (Interpolation [36]) *Given a formula α over the set of actions Σ_1 and another set of actions Σ_2, there is a formula γ over $\Sigma_1 \cap \Sigma_2$ such that $\alpha \Rightarrow \gamma$ is valid and for every formula β over the set of actions Σ_2, if $\alpha \Rightarrow \beta$ is valid then $\gamma \Rightarrow \beta$ is valid.*

The construction of γ as required in the theorem is very straightforward once we have a disjunctive automaton for α (Theorem 12). For every action a not appearing in $\Sigma_1 \cap \Sigma_2$ it is simply enough to replace every formula $(a \to \Gamma)$ by *tt* if the formula is satisfiable and by *ff* otherwise.

26.3.7 Division Property

We want to present one more interesting closure property of the μ-calculus. The motivation comes from modular verification and synthesis. As in the previous subsection, we will consider μ-calculus formulas with only two propositional letters: *tt* and *ff*. This restriction simplifies the notion of a product of transition systems. In a product $\mathcal{M} \times \mathcal{N}$ the states are pairs (s_m, s_n) of states of the two systems. We have a transition from (s_m, s_n) to (s'_m, s'_n) on a letter a iff there is one from s_m to s'_m and one from s_n to s'_n. As we do not have propositions we do not need to say which propositions hold in (s_m, s_n).

Imagine that we fix a transition system $\mathcal{M}$ together with a formula α and face the task: given a transition system $\mathcal{N}$ verify whether $\mathcal{M} \times \mathcal{N} \models \alpha$. That is, verify whether the product of a fixed system with a system given as input satisfies the fixed formula. A straightforward way to solve this problem is to construct the product $\mathcal{M} \times \mathcal{N}$ and then apply a model-checking algorithm. It is possible though to do some pre-processing and construct a formula $\alpha/_{\mathcal{M}}$ as stated in the following theorem.

Theorem 19 (Division Operation [4]) *For every transition system $\mathcal{M}$ and every μ-calculus formula α, there is a μ-calculus formula $\alpha/_{\mathcal{M}}$ such that for every transition system $\mathcal{N}$:*

$$\mathcal{N} \models \alpha/_{\mathcal{M}} \quad \textit{iff} \quad \mathcal{M} \times \mathcal{N} \models \alpha.$$

The construction of $\alpha/_{\mathcal{M}}$ is particularly short using the formalism of modal automata (Sect. 26.3.2). So below we will talk about $\mathcal{A}/_{\mathcal{M}}$ instead of $\alpha/_{\mathcal{M}}$. Consider a modal automaton $\mathcal{A} = \langle Q, \Sigma, q_0, \delta : Q \to \mathcal{F}(\Sigma, Q), \Omega\rangle$ and a transition system $\mathcal{M} = \langle S, \{R_a\}_{a\in\Sigma}\rangle$ both over the same set of actions. In both we just have propositions *tt* and *ff*; this justifies writing $\mathcal{F}(\Sigma, Q)$ instead of $\mathcal{F}(\Sigma, \{tt, ff\}, Q)$. Our goal is to construct an automaton $\mathcal{A}/_{\mathcal{M}}$.

We first define a division α/s for α a formula from $\mathcal{F}(\Sigma, Q)$, and s a state of $\mathcal{M}$. The result is a formula from $\mathcal{F}(\Sigma, Q \times S)$:

$$q/s = (q, s)$$

$$(\alpha \vee \beta)/s = \alpha/s \vee \beta/s \qquad (\alpha \wedge \beta)/s = \alpha/s \wedge \beta/s$$

$$\big(\langle a\rangle\alpha\big)/s = \langle a\rangle \bigvee \big\{\alpha/s' : s \xrightarrow{a} s'\big\} \qquad \big([a]\alpha\big)/s = [a] \bigwedge \big\{\alpha/s' : s \xrightarrow{a} s'\big\}$$

The set of states of $\mathcal{A}/_{\mathcal{M}}$ will be $Q_{/} = Q \times S$. The rank function will be inherited from $\mathcal{A}$: $\Omega_{/}(q, s) = \Omega(q)$. Finally, the transition function will be defined using the above operation: $\delta_{/}(q, s) = \delta(q)/s$. Recall that $\delta(q) \in \mathcal{F}(\mathcal{S}, Q)$, so $\delta(q)/s \in \mathcal{F}(\mathcal{S}, Q \times S)$ as required. Once again, the semantics in terms of games gives the tools to prove correctness of the construction. One can show that Eve can win in $\mathcal{G}(\mathcal{N}, \mathcal{A}/\mathcal{M})$ iff she can win in $\mathcal{G}(\mathcal{M} \times \mathcal{N}, \mathcal{A})$.

The division operation lets us solve some kinds of synthesis problems. Suppose that we are given a finite transition system $\mathcal{M}$ modeling behaviors of a device. We want to restrict these behaviors so that the result satisfies a specification given by a formula α. The restriction is modeled by taking the product of $\mathcal{M}$ with another transition system $\mathcal{C}$. This is a restriction in the sense that every path in $\mathcal{M} \times \mathcal{C}$ is a path in $\mathcal{M}$. Transition system $\mathcal{C}$ is considered to be a controller for $\mathcal{M}$. One may also want to put some constraints on $\mathcal{C}$. For example one can single out a set of uncontrollable actions and demand that these actions cannot be restricted by $\mathcal{C}$. This constraint translates to the requirement that from every state of $\mathcal{C}$ there is a transition on every unobservable action. Hence, this condition can be written as a μ-calculus formula β_{unc}. The synthesis problem then becomes: given $\mathcal{M}$ and α find $\mathcal{C}$ such that

$$\mathcal{M} \times \mathcal{C} \vDash \alpha \quad \text{and} \quad \mathcal{C} \vDash \beta_{unc}.$$

Thanks to Theorem 19 the latter is equivalent to

$$\mathcal{C} \vDash (\alpha/_{\mathcal{M}}) \wedge \beta_{unc}.$$

Thus finding a controller $\mathcal{C}$ is reduced to the satisfiability problem for the μ-calculus (Theorem 13). Let us remark that the Church synthesis problem [32, 116] for μ-calculus formulas is an instance of the above for a particular choice of $\mathcal{M}$.

Not all important constraints though can be expressed in the μ-calculus. For example one can ask that some actions of the system are invisible to a controller. This translates to the requirement that in $\mathcal{C}$ transitions on these actions should be self-loops. This requirement is not invariant under bisimulation, and in consequence is not expressible in the μ-calculus (Theorem 10). Fortunately, it turns out that similar

constructions to the above work for the μ-calculus with the self-loop predicate [9], and that the extended logic is still decidable in EXPTIME.

26.4 Relations with Other Logics

In this section we look at the μ-calculus in a wider context. Monadic second-order logic (MSOL) is a reference for all temporal and program logics because it is a classical formalism capturing recognizability on words and trees [113]. We will explain why the μ-calculus is closely related to MSOL, and why it is in some sense the strongest behavioral logic included in MSOL. It is no surprise then that other well-known temporal and program logics can be translated into the μ-calculus. We briefly present the translations of some of them, and discuss their properties. In order to give a broader picture, we briefly describe two interesting extensions of the μ-calculus. One has a semantical flavor: we add some structure to successors of a node in a transition system. The other is more syntactic: we add a fragment of first-order logic to the μ-calculus.

26.4.1 MSOL, Binary Trees and Bisimulation Invariance

A transition system $\mathcal{M} = \langle S, \{R_a\}_{a\in Act}, \{P_i\}_{i\in\mathbb{N}}\rangle$ can be considered as a model of first-order logic, over the signature consisting of binary relations R_a and unary relations P_i. This logic is not sufficiently expressive for verification as it cannot express such a fundamental property as reachabilty: there is a path from x to y. For this and many other reasons [113] it is interesting to consider *monadic second-order logic*, MSOL. This is an extension of the first-order logic with set variables, denoted $X, Y, \ldots$, the membership predicate $y \in X$, and quantification using set variables $\exists X.\varphi$. For example the formula

$$\forall X.\, \big[(y \in X) \wedge \forall z, z'.\, \big(z \in X \wedge R_b\big(z, z'\big)\big) \Rightarrow z' \in X\big] \Rightarrow \big(y' \in X\big),$$

expresses that y' is reachable from y by a sequence of b actions. The formula literally says that for every set X, if y is in X and X is closed under taking successors with respect to b actions, then y' is in X.

We write $\mathcal{M}, V \vDash \varphi$ to say that an MSOL formula φ holds in the transition system $\mathcal{M}$ under valuation V. Observe that V assigns states of $\mathcal{M}$ to first-order variables in φ and sets of states of $\mathcal{M}$ to set variables in φ. If s is a state of $\mathcal{M}$, and $\varphi(x)$ is a formula with unique free first-order variable x then we simply write $\mathcal{M}, V \vDash \varphi(s)$ for $\mathcal{M}, V[x \mapsto s] \vDash \varphi(x)$, i.e., to say that $\varphi(x)$ is true under a valuation that maps x to s.

Since the satisfiability problem for first-order logic over transition systems is undecidable, so is the one for MSOL. The situation is much more interesting for tree transition systems. Rabin's theorem [98, 113] says that MSOL over tree transition

systems is decidable. So it is natural to try to compare its expressive power with that of the μ-calculus over trees. To this end we need to find a common ground for the two logics. A small problem we need to overcome is that MSOL talks about the truth in a tree, while the μ-calculus talks about the truth in a node of the tree. Then also we need to take care of first-order variables that are allowed in MSOL, but not in the μ-calculus. We will discuss these points in more detail.

It is quite straightforward to translate the μ-calculus to MSOL. The translation goes by induction on the structure of the formula. For a formula α of the μ-calculus we construct a formula $\varphi(x)$ of MSOL with the same second-order variables and one free first-order variable x. The translation has the property that for every transition system $\mathcal{M}$, its state s, and valuation of second-order variables V, we have:

$$\mathcal{M}, s, V \vDash \alpha \quad \text{iff} \quad \mathcal{M}, V \vDash \varphi(s). \tag{3}$$

For example, we translate a variable Y of the μ-calculus to a formula $x \in Y$. Similarly, a proposition p is translated to $P(x)$. We translate Boolean connectives to the same Boolean connectives. For modalities and fixpoints we express semantic clauses from Sect. 26.2.1 in MSOL.

The translation in the other direction is not so obvious. First of all, we consider formulas with only one free first-order variable x. So below when we talk about MSOL we consider only such formulas. We say that such a formula is *equivalent* to a μ-calculus formula α if the equivalence (3) holds. Observe that MSOL can express properties that are not bisimulation invariant, for example "a node has three successors" or "there is a cycle". Since the μ-calculus can express only bisimulation invariant properties (Theorem 10), it cannot be equivalent to MSOL over transition systems. This observation justifies the restriction to deterministic tree transition systems where every successor has a unique name, and there are no cycles.

Theorem 20 (Equivalence with MSOL [90]) *Over deterministic tree transition systems MSOL is equivalent to the μ-calculus.*

An interesting variation of MSOL, called *weak-MSOL*, is obtained by restricting the set variables to range over finite sets only. Another interesting class is that of Σ_1 MSOL formulas, which have a prefix of (unrestricted) existential quantifiers followed by a formula without second-order quantifiers. Rabin has shown [99] that an MSOL formula is equivalent to a weak-MSOL formula iff both the formula and its negation are equivalent to Σ_1 MSOL formulas. He has also shown that Σ_1 MSOL formulas are equivalent to tree automata with Büchi acceptance conditions. It turns out that weak-MSOL is equivalent to the alternation-free fragment of the μ-calculus (which we recall from Sect. 26.2.2 is the fragment consisting of formulas of alternation depth 1).

Theorem 21 (Equivalence with Weak-MSOL [90]) *Over deterministic tree transition systems, weak-MSOL is equivalent to the alternation-free μ-calculus. The Σ_1 fragment of MSOL is equivalent to the Π_2^μ fragment of the μ-calculus.*

If we drop the determinacy restriction then there is a simple extension of the μ-calculus that captures MSOL on tree transition systems. It is enough to introduce counting modalities $\langle b\rangle_{=n}\alpha$ for every action b and natural number n. The meaning of this modality is that there are exactly n distinct b-successors of a node satisfying α. This observation is an instance of the more general framework described in Sect. 26.4.3.1.

Let us come back to the case of transition systems with no restrictions. We have noted above that if an MSOL formula $\varphi(x)$ is equivalent to a μ-calculus formula then it is *bisimulation invariant*: if a formula holds in a state then it holds in every state bisimilar to it. It turns out that the converse implication holds.

Theorem 22 (Expressive Completeness [69]) *The μ-calculus is expressively equivalent to the bisimulation invariant fragment of MSOL: if an MSOL formula $\varphi(x)$ is bisimulation invariant then it is equivalent to a μ-calculus formula.*

Recall that the satisfiability problem for MSOL over transition systems is not decidable [41]. In consequence, it is not decidable if a given MSOL formula is bisimulation invariant. So it is not decidable whether an MSOL formula can be written in the μ-calculus.

26.4.2 Embedding of Program Logics

The μ-calculus is one of the numerous program logics designed to express properties of transition systems. Theorem 22 implies that the μ-calculus is as expressive as any logic that is at the same time bisimulation invariant and not more expressive than MSOL. This covers most program logics, for example, propositional dynamic logic (PDL), computational tree logic (CTL), and its extension CTL* [43, 50]. It is anyway worthwhile to see explicit translations of these logics into the μ-calculus and discuss their relative expressive powers.

Translating PDL or CTL into the μ-calculus is easy. For an example let us look at CTL. The formulas of this logic are built from *tt* and *ff* using Boolean connectives, the $\langle a\rangle$ modality, and two operators:

$$\mathsf{E}(\alpha\mathsf{U}\beta) \qquad \mathsf{E}\neg(\alpha\mathsf{U}\beta).$$

As for the μ-calculus, the meaning of a CTL formula is a set of states. The only construct with non-obvious semantics is the until operator:

$\mathcal{M}, s \vDash \mathsf{E}(\alpha\mathsf{U}\beta)$ iff there is a path $s_0, s_1, \ldots, s_k$ such that $s_0 = s$, $s_k \vDash \beta$, and $s_i \vDash \alpha$ for $i = 0, \ldots, k-1$.

For the translation into the μ-calculus, denoted $[\alpha]^\natural$, we just need to take care of the two new operators. We show the translation only for the first one:

$$\big[\mathsf{E}(\alpha\mathsf{U}\beta)\big]^\natural \quad \text{is} \quad \mu X.[\beta]^\natural \vee \left([\alpha]^\natural \wedge \bigvee_{a\in\Sigma} \langle a\rangle X\right).$$

The translation produces an alternation-free formula and is linear in size. The translation for PDL is equally easy, but to get linear-size formulas we need to use the vectorial syntax of the μ-calculus (Sect. 26.2.1).

Translating linear-time logics, like LTL, is much more complicated. First one needs to make LTL express properties of transition systems. This can be done easily by saying that a formula is true in a state if it is true on all maximal paths from that state. For example, a formula is true in a structure with one state and self-loops on every action iff it is true on every infinite path. Since validity of LTL formulas is PSPACE-complete, this indicates that one can expect an exponential blowup when translating from LTL to the μ-calculus. It is easy to write a μ-calculus formula that is equivalent to LTL over linear models, i.e., transition systems consisting of only one path. For the general case, the problem is that the μ-calculus does not have an explicit path quantifier, and there is no easy way to say "every path satisfies some formula". The solution for translating LTL is more complicated. One translates an LTL formula into a Büchi automaton accepting the sequences not satisfying the formula ([54] or [76] in this Handbook). Then this automaton is translated into a μ-calculus formula over linear models. It can be shown that such a formula can be directly used to express the property that there is a path in the transition system not satisfying the initial LTL formula. The negation of this formula is the desired translation. Since CTL* is a common extension of both CTL and LTL, its translation into the μ-calculus combines the two translations described above [16, 38]. One can show that CTL* is contained in the Σ_3^μ level of the alternation hierarchy. In the translation described above one can use nondeterministic Büchi automata instead of parity automata. This gives a Π_2^μ formula for every path quantifier. Due to negation and other constructs of CTL*, the complete translation gives a combination of Π_2^μ and Σ_2^μ formulas.

Fact 23 (Embedding of Program Logics) *Every formula of CTL, PDL, or CTL* can be translated into an equivalent μ-calculus formula.*

Concerning expressiveness, CTL is quite a weak logic. It cannot express for example that there is a path with infinitely many b events on it. This follows from the strictness of the μ-calculus hierarchy (Theorem 16) as CTL can be translated into the alternation-free μ-calculus. Even though CTL* can express the "infinitely many" property, it is still expressively weaker than the μ-calculus. For example, it cannot express properties describing infinite interaction of the type "there is a way of repeatedly choosing an output for a given input so that the resulting infinite sequence of inputs and outputs satisfies some given LTL property". In particular, CTL* cannot express all game formulas (1) as these formulas can express properties arbitrarily high in the alternation hierarchy (Sect. 26.3.4).

The last remark provided an inspiration for the introduction of alternating-time temporal logic [2]. It has been defined over a large class of so-called game models, so it is not immediately comparable to the μ-calculus. Yet, in the cases when alternating-time temporal logic behaves well, the logic can be translated into the μ-calculus. Another interesting logic is game logic [93]. This logic can be translated

into the μ-calculus with only two variables, yet it can express properties arbitrarily high in the alternation hierarchy [12].

26.4.3 Beyond μ-Calculus

Because of its simple formulation, its expressive power, and its algorithmic properties, the μ-calculus has proved to be a very valuable logic for verification. Like every system, the logic has its limitations, and some of them we have already mentioned in this chapter. The μ-calculus does not allow any form of equality: we cannot say that a transition is a self-loop, or that transitions on a and b go to the same state. It permits no form of counting: we cannot say that there is only one a-transition from a state. It does not have quantification: we cannot say that a formula holds in every state. It does not allow derived transition relations: we cannot talk about the reverse of a transition relation. Obviously, such a wish list has no end, but it provides good motivation in a search for extensions of the μ-calculus. Of course it is not very difficult to define very expressive logics. What is important is to find extensions without losing good properties of the logic, or at least not all of them.

In this subsection we will present two extensions of the μ-calculus of different natures. The first starts from model-theoretic ideas, the second is motivated by the syntax. The two extensions are quite different in what they accomplish, but both have proved to be very useful.

26.4.3.1 Iterated Structures

Till now we have considered logics over transition systems. Here we would like to consider what happens when we put some structure on the set of successors of a state of a transition system. For example, what would happen if we added an order among successors. Instead of considering particular cases we will introduce a general method of constructing transition systems with additional structure. Then we will discuss how to handle this added structure in extensions of the μ-calculus. For this we will define automata that are a generalization of modal automata. We will present their closure properties as well as their relation to MSOL.

Let $\mathcal{N} = \langle D, r_1, \dots \rangle$ be a structure over a relational signature; this means that functional symbols are not allowed. For example, $\langle \mathbb{N}, \leq \rangle$ is a structure over a signature containing one relation, the standard ordering; the structure $\langle \mathbb{N}, + \rangle$ is allowed when $+$ is considered as a ternary addition relation, and not as a binary function.

An *iterated structure* is a structure $\mathcal{N}^* = \langle D^*, \mathit{child}, \mathit{clone}, r_1^*, \dots \rangle$ where D^* is the set of all finite sequences over D, and the relations are defined by:

$$\begin{aligned} \mathit{child} &= \big\{(w, wd) : w \in D^*,\ d \in D\big\}, \\ r_i^* &= \big\{(wd_1, \dots, wd_k) : w \in D^*,\ (d_1, \dots, d_k) \in r_i\big\}, \\ \mathit{clone} &= \big\{wdd : w \in D^*,\ d \in D\big\}. \end{aligned}$$

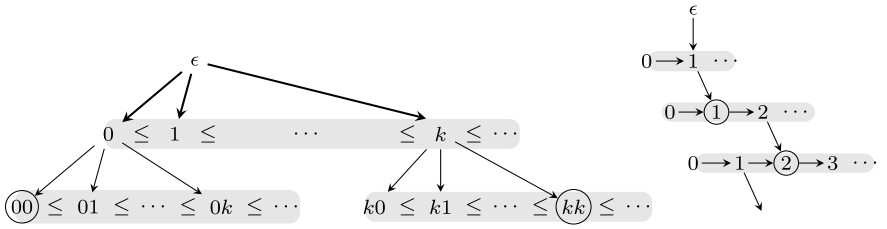


Fig. 7 Structure $\mathcal{N}_{\leq}^*$, and a representation of the sequence $1, 2, 1, \ldots$ in $\mathcal{N}_{\leq}^*$

So $\mathcal{N}^*$ is a tree where every node has a rank equal to the size of D; indeed a node $w \in D^*$ has successors wd for every $d \in D$. The relation *child* gives the successors of a node. The relations r_i^* induce additional dependencies among siblings that come from the initial structure. The unary relation *clone* is a curious predicate saying that a node ends in the same symbol as its parent. This predicate is very useful in encoding of other structures into iterated structures.

Example: Consider a two element structure $\mathcal{N}_2 = \langle \{0, 1\}, \mathit{left} \rangle$ with one unary relation that holds only for 0. The structure $\mathcal{N}_2^*$ has elements $\{0, 1\}^*$. The relation $\mathit{left}^*(w)$ holds if w ends with 0. Hence $\mathcal{N}_2^*$ is the full binary tree with left^* designating left sons. The *clone* relation is not important here: it holds in a left child whose parent is also a left child (or analogously for right).

Example: Consider the set of natural numbers with the standard ordering: $\mathcal{N}_{\leq} = \langle \mathbb{N}, \leq \rangle$. In this case $\mathcal{N}_{\leq}^*$ is a tree of infinite branching where siblings are linearly ordered. Moreover, the *clone* predicate designates a node whose position among its siblings is the same as that of the father. The structure is shown in Fig. 7, where the circled nodes satisfy the clone predicate. The clone predicate allows us to define in this tree paths representing a possible behavior of one register with $+1$ and -1 operations: paths of the form $i_1 i_2 \ldots$ with $i_1 = 0$ and $|i_{k+1} - i_k| \leq 1$. The idea is presented in Fig. 7. The clone predicate allows us to go one step to the left or one step to the right in the horizontal linear order when going one level down in the tree.

In general, for a given structure $\mathcal{N} = \langle D, r_1, \ldots \rangle$, the structure $\mathcal{N}^*$ can be seen as a generalization of the notion of the full binary tree. Given a finite alphabet Σ, a Σ-labeling of $\mathcal{N}$ is just a function $t : \mathcal{N}^* \to \Sigma$. We are going to define the notion of an automaton that accepts sets of labeled iterated structures.

First, let $\mathcal{MF}(\Sigma, Q)$ be the set of monadic second-order formulas with free variables $\{X_{\mathit{clone}}\} \cup \{X_q : q \in Q\}$ over the signature of $\mathcal{N}$ enriched with monadic predicates P_a for $a \in \Sigma$. The automaton is:

$$\mathcal{A} = \left\langle Q, \Sigma, q^0 \in Q, \delta : Q \to \mathcal{MF}(\Sigma, Q), \Omega : Q \to \mathbb{N} \right\rangle$$

The *acceptance* of $t : \mathcal{N}^* \to \Sigma$ by $\mathcal{A}$ is defined in terms of the *acceptance game* $\mathcal{G}(t, \mathcal{A})$.

- The positions for Eve are $D^* \times Q$, the positions for Adam are $D^* \times (Q \to \mathcal{P}(D))$. The initial position is (ε, q^0). Recall that D^* is the set of elements of $\mathcal{N}^*$, in particular ε is the root of $\mathcal{N}^*$.
- In a position (w, q) Eve chooses a function $f : Q \to \mathcal{P}(D)$ so that $\mathcal{N}, \mathcal{V} \vDash \delta(q)$ where $\mathcal{V}(X_q) = f(q)$ and $\mathcal{V}(X_{clone})$ is the set consisting of the last element of w, or $\varnothing$ if w has length ≤ 1. The game moves to the position (w, f).
- In a position (w, f) Adam chooses $q' \in Q$ and $d' \in f(q')$. The game moves to (wd', q').
- The rank of a position (w, q) is $\Omega(q)$. All the positions for Adam have rank 0.

A labeled iterated structure is *accepted* if Eve has a winning strategy in this game from the initial position. Hence an automaton accepts a set of labeled iterated structures, for a fixed initial structure. The following theorem was formulated by A. Muchnik. The sketch of his proof can be found in [109]. The result is proved with a different proof method in [124].

Theorem 24 (Transfer Theorem [109, 124]) *Let $\mathcal{N}$ be a relational structure. Automata over labeled $\mathcal{N}$-iterated structures are closed under Boolean operations and projection. If the MSOL theory of $\mathcal{N}$ is decidable then the emptiness of automata over labeled $\mathcal{N}$-iterated structures is decidable.*

In particular, when $\mathcal{N}_2$ is the two-element structure from the example above, Theorem 24 gives decidability of the MSOL theory of the binary tree. Observe that modal automata (Sect. 26.3.2) are a special case of automata from this section for the structures of the empty signature. We have seen that modal automata and the μ-calculus are essentially the same. Following the same ideas one can construct μ-calculi for different kinds of relations on the set of successor states.

Structure iteration is a powerful operation preserving decidability of MSOL theories, that is, it transforms a structure with decidable MSOL theory into another structure with decidable MSOL theory. Among other things, it is an entry point to the so-called pushdown hierarchy, model-checking higher-order pushdown systems, and higher-order recursive schemes. This is a vast subject that would require a chapter on its own [29, 30, 73, 74, 92, 115].

26.4.3.2 Guarded Logics

The idea of guarded logics comes from looking at the translation of the modal logic into first-order logic. In this translation the Boolean connectives are translated to themselves, and the modalities are translated as follows:

$$\big[\langle a \rangle \alpha\big]^*(x) \quad \text{is} \quad \exists y.\ R_a(x, y) \wedge [\alpha]^*(y)$$

$$\big[[a]\alpha\big]^*(x) \quad \text{is} \quad \forall y.\ R_a(x, y) \Rightarrow [\alpha]^*(y).$$

As for MSOL, the translation is parameterized by a free variable intended to stand for the current state.

The image of this translation is called the *modal fragment* of first-order logic. Recall that the satisfiability problem for first-order logic over transition systems is undecidable. But it is decidable for the modal fragment since it is decidable for modal logic. This immediately brings up the questions: what makes the modal fragment so special, and can it be extended? The idea behind the guarded fragment is to provide an answer to this question by focusing on the restricted use of quantifiers.

We say that a quantification is *guarded* if it is of the form

$$\exists y.\gamma(x,y) \wedge \psi(x,y) \quad \text{or} \quad \forall y.\gamma(x,y) \Rightarrow \psi(x,y),$$

where $\gamma(x,y)$ is $R_b(x,y)$ or $R_b(y,x)$ for some action b, and $\psi(x,y)$ is a formula whose free variables are at most x and y. The name "guarded" comes from the fact that the quantified variable has to be related to the free variable by the transition relation. For example the formula

$$\forall y.R_a(x,y) \Rightarrow x = y$$

says that all transitions on a are self-loops. The syntax also permits one to talk directly about the reverse of transitions.

The *guarded fragment* [5] is the set of formulas of first-order logic that contains atomic formulas, and is closed under Boolean operations and guarded quantification. The presentation here is limited to the simplest variant. The fragment gets even more interesting for signatures with relations of higher arity.

Since the guarded fragment inherits many good properties of modal logic, its extension with fixpoints should inherit those of the μ-calculus. This is indeed the case. If one takes some care as to how fixpoints are applied, one can even recover many good properties of the μ-calculus.

Let W be a unary relation variable, let $\psi(W,x)$ be a guarded formula whose free variables are as displayed, and where W appears only positively and not in guards. Then we can build formulas:

$$[\mathsf{lfp}\ Wx.\ \psi](x) \quad \text{and} \quad [\mathsf{gfp}\ Wx.\ \psi](x).$$

The semantics is as expected: The formula $[\mathsf{lfp}\ Wx.\psi](s)$ is true in a transition system $\mathcal{M}$ iff s is in the least fixpoint of the operator mapping a set of states S' to $\{s' : \mathcal{M} \models \psi(S', s')\}$, i.e., to the set of states s' that satisfy $\psi(W,x)$ when W is interpreted by S'. The extension of the guarded logic with these two constructs is called *guarded fixpoint logic*.

Let us see an example formula of guarded fixpoint logic that is not equivalent to a μ-calculus formula:

$$\begin{aligned}&\big(\exists xy.\ R_b(x,y)\big) \wedge \big(\forall xy.\ R_b(x,y) \Rightarrow \exists z.\ R_b(y,z)\big)\\ &\quad \wedge \forall xy.\ R_b(x,y) \Rightarrow \big[\mathsf{lfp}\ Wz.\ \forall y.\ R_b(y,z) \Rightarrow W(y)\big](x).\end{aligned}$$

The first two conjuncts say that there is a transition labeled by b, and that every such transition can be extended to an infinite path. The third conjunct says that

every source of a b-transition has only finitely many predecessors on b transitions. Thus the formula implies that there is an infinite forward chain of b-transitions but no infinite backward chain. This example shows the use of backwards modalities and the price to pay for them: we can write formulas having only infinite models.

Despite this observation the satisfiability problem for the guarded fragment extended with fixpoints is decidable, and the complexity is the same as for the μ-calculus.

Theorem 25 (Guarded Fixpoint Logic [60]) *The satisfiability problem for guarded fixpoint logic over transition systems is* EXPTIME*-complete.*

Adaptations of many results presented in this chapter hold for guarded fixpoint logic: game characterization of model checking, a tree-model property, bisimulation invariance, and iteration of structures [20, 56, 57].

26.5 Related Work

As with any good concept, the μ-calculus can be approached from many directions. The first point of view is to consider it as a logic of programs. The family of logics of programs is divided into two groups. In exogenous logics, a program is a part of a formula; in endogenous logics, a program is a part of a model. Dynamic logic and Hoare logic are examples of exogenous logics. Temporal logic and Floyd diagrams are endogenous logics. The μ-calculus also belongs to this second group.

Exogenous logics merit a small digression. Among them, dynamic logics are the closest to the μ-calculus. Historically, the research on dynamic logics has been an intermediate step to major results of the theory of the μ-calculus [110, 111]. Dynamic logics were developed independently by Slawicki [101] and Pratt [97]. A propositional version of dynamic logic was proposed and studied by Fisher and Ladner [49]. A survey of Harel gives a very good overview of the subject [61]. A more recent reference is the book of Harel, Kozen and Tiuryn [62].

The second point of view is that the μ-calculus is a propositional version of the least fixpoint logic: an extension of first-order logic with fixpoint operators. From this point of view Y. Moschovakis' work in model theory laid foundations for the logic [85]. Least fixpoint logic and the closely related inflationary fixpoint logic are intensively studied in finite-model theory [40, 58, 66, 77]. Inflationary fixpoint logic has its propositional version too [39]; while it has greater expressive power, it is less algorithmically manageable than the μ-calculus.

The third point of view is to consider the μ-calculus as a basic modal logic extended with a fixpoint operator. Modal logic was proposed by philosophers at the beginning of the twentieth century [18]. In the 1950s a possible worlds semantics was introduced, and since then modal logic has proven to be an appealing language to describe properties of transition systems. A number of ways of using a fixpoint operator in program logics have been proposed [64, 94, 97, 105]. The μ-calculus as it is known now was formulated by Kozen in [75].

The expressive completeness theorem for the μ-calculus, Theorem 22, is an analog of van Benthem's theorem [117] saying that a first-order formula is equivalent to a modal formula if and only if it is bisimulation invariant. So Theorem 22 says that μ-calculus is to MSOL what modal logic is to first-order logic. Expressive completeness results have also been proved for extensions of the μ-calculus as well as for its fragments [57, 67].

As stated in Theorem 8, the model-checking problem is equivalent to checking the emptiness of nondeterministic parity automata on infinite trees. Similarly, the satisfiability problem is linked to automata emptiness (see Theorem 14). Many arguments in this chapter have an automata-theoretic flavor. This explains the relevance of the study of determinization and complementation operations for automata for the theory of the μ-calculus [34, 95, 100, 104, 119].

As with any successful formalism it is very tempting to extend the μ-calculus while retaining most of its good properties. In some sense the expressive completeness theorem tells us that it is not possible to keep all the good properties. In Sect. 26.4.3.2 we have described guarded fixpoint logics. The μ-calculi with backwards modalities, loop modalities, etc. have been studied separately [9, 25, 118]. There exists also a μ-calculus for timed transition systems [26, 63]. Quantitative versions of the μ-calculus have also been proposed: be it for probabilistic transition systems [1, 55, 81, 84], or for some form of discounting [48]. In Sect. 26.4.3.1 we have seen how to extend μ-calculus to the case when the set of successors of a state has some structure, for example linear order. Another extension in a similar spirit is the coalgebraic μ-calculus [121].

The results on the model-checking problem have been discussed in Sect. 26.2.4. It is worth mentioning that the problem has also been studied on some special classes of transition systems: bounded tree and clique-width [91], bounded entanglement [13], and undirected graphs [15, 37].

The alternation hierarchy discussed in Sect. 26.3.4 is the most established way of stratifying μ-calculus properties. Another hierarchy is the variable hierarchy obtained by limiting the number of variables that can be used to write a formula. Most common program logics, CTL, PDL, CTL*, and even the game logic of Parikh [93] are contained in the first two levels of this hierarchy. Still the variable hierarchy is strict [14].

As we have seen in Sect. 26.4.2, most program logics such as CTL* or PDL can be translated into the μ-calculus. It would be interesting to understand which formulas of the μ-calculus correspond to formulas of, say, CTL. Put differently, for a logic L we would like to decide whether a given μ-calculus formula is equivalent to a formula of L. Decidability of this problem is known for several fragments of first-order logic [11, 21–24, 96]. However, the problem is open for all major program logics like CTL, CTL*, or PDL.

There exist two other surveys on the subject that present the logic from different angles [28, 120]. For coverage in depth of the theory of the μ-calculus we refer the reader to the book of Arnold and Niwiński [8].

References

1. de Alfaro, L., Majumdar, R.: Quantitative solution of omega-regular games. J. Comput. Syst. Sci. **68**(2), 374–397 (2004)
2. Alur, R., Henzinger, T., Kupferman, O.: Alternating-time temporal logic. In: Int. Symp. on Foundations of Computer Science, pp. 100–109. IEEE, Piscataway (1997)
3. Andersen, H.: Model checking boolean graphs. Theor. Comput. Sci. **126**(1), 3–30 (1994)
4. Andersen, H.R.: Partial model checking. In: Ann. Symp. on Logic in Computer Science, pp. 398–407. IEEE, Piscataway (1995)
5. Andréka, H., van Benthem, J., Neméti, I.: Modal logics and bounded fragments of predicate logic. J. Philos. Log. **27**, 217–274 (1998)
6. Arnold, A.: The mu-calculus alternation-depth hierarchy is strict on binary trees. RAIRO Theor. Inform. Appl. **33**, 329–339 (1999)
7. Arnold, A., Crubille, P.: A linear time algorithm to solve fixpoint equations on transition systems. Inf. Process. Lett. **29**, 57–66 (1988)
8. Arnold, A., Niwiński, D.: Rudiments of μ-Calculus. Elsevier, Amsterdam (2001)
9. Arnold, A., Walukiewicz, I.: Nondeterministic controllers of nondeterministic processes. In: Flum, J., Grädel, E., Wilke, T. (eds.) Logic and Automata, Texts in Logic and Games, vol. 2, pp. 29–52. Amsterdam University Press, Amsterdam (2007)
10. Bekic, H.: Definable operation in general algebras, and the theory of automata and flowcharts. In: Jones, C.B. (ed.) Programming Languages and Their Definition—Hans Bekic (1936–1982). LNCS, vol. 177, pp. 30–55. Springer, Heidelberg (1984)
11. Benedikt, M., Segoufin, L.: Regular tree languages definable in FO and in FO_{mod}. Trans. Comput. Log. **11**(1), 4:1–4:32 (2009)
12. Berwanger, D.: Game logic is strong enough for parity games. Stud. Log. **75**(2), 205–219 (2003)
13. Berwanger, D., Grädel, E., Kaiser, L., Rabinovich, R.: Entanglement and the complexity of directed graphs. Theor. Comput. Sci. **463**, 2–25 (2012)
14. Berwanger, D., Grädel, E., Lenzi, G.: The variable hierarchy of the mu-calculus is strict. Theory Comput. Syst. **40**(4), 437–466 (2007)
15. Berwanger, D., Serre, O.: Parity games on undirected graphs. Inf. Process. Lett. **112**(23), 928–932 (2012)
16. Bhat, G., Cleaveland, R.: Efficient model checking via the equational μ-calculus. In: Clarke, E.M. (ed.) Ann. Symp. on Logic in Computer Science, pp. 304–312. IEEE, Piscataway (1996)
17. Björklund, H., Vorobyov, S.G.: A combinatorial strongly subexponential strategy improvement algorithm for mean payoff games. Discrete Appl. Math. **155**(2), 210–229 (2007)
18. Blackburn, R., de Rijke, M., Venema, Y.: Modal Logic. Cambridge University Press, Cambridge (2001)
19. Bloem, R., Chatterjee, K., Jobstmann, B.: Graph games and reactive synthesis. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
20. Blumensath, A., Kreutzer, S.: An extension to Muchnik's theorem. J. Log. Comput. **13**, 59–74 (2005)
21. Bojanczyk, M., Segoufin, L., Straubing, H.: Piecewise testable tree languages. Log. Methods Comput. Sci. **8**(3) (2012)
22. Bojanczyk, M., Straubing, H., Walukiewicz, I.: Wreath products of forest algebras, with applications to tree logics. Log. Methods Comput. Sci. **3**, 19 (2012)
23. Bojanczyk, M., Walukiewicz, I.: Characterizing EF and EX tree logics. Theor. Comput. Sci. **358**(2–3), 255–272 (2006)
24. Bojanczyk, M., Walukiewicz, I.: Forest algebras. In: Flum, J., Grädel, E., Wilke, T. (eds.) Logic and Automata, Texts in Logic and Games, vol. 2, pp. 107–132. Amsterdam University Press, Amsterdam (2007)

25. Bonatti, P.A., Lutz, C., Murano, A., Vardi, M.Y.: The complexity of enriched mu-calculi. Log. Methods Comput. Sci. **3**, 11 (2008)
26. Bouyer, P., Cassez, F., Laroussinie, F.: Timed modal logics for real-time systems: specification, verification and control. J. Log. Lang. Inf. **20**, 169–203 (2011)
27. Bradfield, J.: The modal mu-calculus alternation hierarchy is strict. Theor. Comput. Sci. **195**, 133–153 (1997)
28. Bradfield, J., Stirling, C.: Modal mu-calculi. In: Blackburn, P., van Benthem, J., Wolter, F. (eds.) The Handbook of Modal Logic, pp. 721–756. Elsevier, Amsterdam (2006)
29. Broadbent, C., Carayol, A., Ong, L., Serre, O.: Recursion schemes and logical reflection. In: Ann. Symp. on Logic in Computer Science, pp. 120–129. IEEE, Piscataway (2010)
30. Carayol, A., Wöhrle, S.: The Caucal hierarchy of infinite graphs in terms of logic and higher-order pushdown automata. In: Pandya, P.K., Radhakrishnan, J. (eds.) Intl. Conf. on Foundations of Software Technology and Theoretical Computer Science. LNCS, vol. 2914, pp. 112–124. Springer, Heidelberg (2003)
31. Chatterjee, K., Henzinger, M.: An $O(n^2)$ time algorithm for alternating Büchi games. In: Indyk, P. (ed.) Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA). SIAM, Philadelphia (2012)
32. Church, A.: Applications of recursive arithmetic to the problem of circuit synthesis. In: Summaries of the Summer Institute of Symbolic Logic, vol. I, pp. 3–50. Cornell University, Ithaca (1957)
33. Cleaveland, R., Steffen, B.: A linear model checking algorithm for the alternation-free modal μ-caluclus. Form. Methods Syst. Des. **2**, 121–147 (1993)
34. Colcombet, T., Zdanowski, K.: A tight lower bound for determinization of transition labeled Büchi automata. In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S., Thomas, W. (eds.) International Colloquium on Automata, Languages and Programming. LNCS, vol. 5556, pp. 151–162. Springer, Heidelberg (2009)
35. Condon, A.: The complexity of stochastic games. Inf. Comput. **96**(2), 203–224 (1992)
36. D'Agostino, G., Hollenberg, M.: Logical questions concerning the mu-calculus: interpolation, Lyndon and Łoś-Tarski. J. Symb. Log. **65**(1), 310–332 (2000)
37. D'Agostino, G., Lenzi, G.: On modal mu-calculus over reflexive symmetric graphs. J. Log. Comput. **23**(3), 445–455 (2013)
38. Dam, M.: CTL* and ECTL* as fragments of the modal μ-calculus. Theor. Comput. Sci. **126**(1), 77–96 (1994)
39. Dawar, A., Grädel, E., Kreutzer, S.: Inflationary fixed points in modal logic. Trans. Comput. Log. **5**(2), 282–315 (2004)
40. Ebbinghaus, H.D., Flum, J.: Finite Model Theory. Springer, Heidelberg (1999)
41. Ebbinghaus, H.D., Flum, J., Thomas, W.: Mathematical Logic. Springer, New York (1984)
42. Emerson, E., Jutla, C., Sistla, A.: On model-checking for the mu-calculus and its fragments. Theor. Comput. Sci. **258**(1–2), 491–522 (2001)
43. Emerson, E.A.: Temporal and modal logic. In: Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, vol. B, pp. 995–1072. Elsevier, Amsterdam (1990)
44. Emerson, E.A., Jutla, C.S.: The complexity of tree automata and logics of programs. In: Int. Symp. on Foundations of Computer Science, pp. 328–337. IEEE, Piscataway (1988)
45. Emerson, E.A., Jutla, C.S.: Tree automata, mu-calculus and determinacy. In: Int. Symp. on Foundations of Computer Science, pp. 368–377. IEEE, Piscataway (1991)
46. Emerson, E.A., Jutla, C.S.: The complexity of tree automata and logics of programs. SIAM J. Comput. **29**(1), 132–158 (1999)
47. Emerson, E.A., Lei, C.: Efficient model checking in fragments of propositional mu-calculus. In: Ann. Symp. on Logic in Computer Science, pp. 267–278. IEEE, Piscataway (1986)
48. Fischer, D., Grädel, E., Kaiser, L.: Model checking games for the quantitative mu-calculus. Theory Comput. Syst. **47**, 696–719 (2010)
49. Fisher, M., Ladner, R.: Propositional modal logic of programs. In: Hopcroft, J.E., Friedman, E.P., Harrison, M.A. (eds.) Annual Symp. on the Theory of Computing, pp. 286–294. ACM, New York (1977)

50. Fisher, M., Ladner, R.: Propositional dynamic logic of regular programs. J. Comput. Syst. Sci. **18**, 194–211 (1979)
51. Friedmann, O.: An exponential lower bound for the latest deterministic strategy iteration algorithms. Log. Methods Comput. Sci. **3**, 23 (2011)
52. Friedmann, O., Hansen, T.D., Zwick, U.: A subexponential lower bound for the random facet algorithm for parity games. In: Randall, D. (ed.) Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 202–216. SIAM, Philadelphia (2011)
53. Friedmann, O., Lange, M.: The modal mu-calculus caught off guard. In: Brünnler, K., Metcalfe, G. (eds.) Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX). LNCS, vol. 6793, pp. 149–163. Springer, Heidelberg (2011)
54. Gastin, P., Oddoux, D.: Fast LTL to Büchi automata translation. In: Berry, G., Comon, H., Finkel, A. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 2102, pp. 53–65. Springer, Heidelberg (2001)
55. Gimbert, H., Zielonka, W.: Perfect information stochastic priority games. In: Arge, L., Cachin, C., Jurdzinski, T., Tarlecki, A. (eds.) International Colloquium on Automata, Languages and Programming. LNCS, vol. 4596, pp. 850–861. Springer, Heidelberg (2007)
56. Grädel, E.: Guarded fixed point logics and the monadic theory of countable trees. Theor. Comput. Sci. **288**(1), 129–152 (2002)
57. Grädel, E., Hirsch, C., Otto, M.: Back and forth between guarded and modal logics. Trans. Comput. Log. **3**(3), 418–463 (2002)
58. Grädel, E., Kolaitis, P., Libkin, L., Marx, M., Spencer, J., Vardi, M., Venema, Y., Weinstein, S.: Finite Model Theory and Its Applications. Springer, Heidelberg (2007)
59. Grädel, E., Thomas, W., Wilke, T. (eds.): Automata, Logics, and Infinite Games, vol. 2500. Springer, Heidelberg (2002)
60. Grädel, E., Walukiewicz, I.: Guarded fixed point logic. In: Ann. Symp. on Logic in Computer Science, pp. 45–55. IEEE, Piscataway (1999)
61. Harel, D.: Dynamic logic. In: Gabbay, D., Guenther, F. (eds.) Handbook of Philosophical Logic, Vol. II, pp. 497–604. Reidel, Dordrecht (1984)
62. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. MIT Press, Cambridge (2000)
63. Henzinger, T.A., Nicollin, X., Sifakis, J., Yovine, S.: Symbolic model checking for real-time systems. Inf. Comput. **111**(2), 193–244 (1994)
64. Hitchcock, P., Park, D.: Induction rules and termination proofs. In: Nivat, M. (ed.) International Colloquium on Automata, Languages and Programming, pp. 225–251 (1973)
65. Hoffman, A., Karp, R.: On nonterminating stochastic games. Manag. Sci. **12**, 359–370 (1966)
66. Immerman, N.: Descriptive Complexity. Springer, Heidelberg (1999)
67. Janin, D., Lenzi, G.: On the relationship between monadic and weak monadic second order logic on arbitrary trees, with applications to the mu-calculus. Fundam. Inform. **61**(3–4), 247–265 (2004)
68. Janin, D., Walukiewicz, I.: Automata for the mu-calculus and related results. In: Wiedermann, J., Hájek, P. (eds.) International Symposium on Mathematical Foundations of Computer Science. LNCS, vol. 969, pp. 552–562. Springer, Heidelberg (1995)
69. Janin, D., Walukiewicz, I.: On the expressive completeness of the propositional mu-calculus with respect to monadic second order logic. In: Montanari, U., Sassone, V. (eds.) Intl. Conf. on Concurrency Theory (CONCUR). LNCS, vol. 1119, pp. 263–277. Springer, Heidelberg (1996)
70. Jurdziński, M.: Deciding the winner in parity games is in UP∩co-UP. Inf. Process. Lett. **68**(3), 119–124 (1998)
71. Jurdziński, M.: Small progress measures for solving parity games. In: Reichel, H., Tison, S. (eds.) Annual Symposium on Theoretical Aspects of Computer Science. LNCS, vol. 1770, pp. 290–301. Springer, Heidelberg (2000)
72. Jurdzinski, M., Paterson, M., Zwick, U.: A deterministic subexponential algorithm for solving parity games. SIAM J. Comput. **38**(4), 1519–1532 (2008)

73. Knapik, T., Niwinski, D., Urzyczyn, P.: Higher-order pushdown trees are easy. In: Nielsen, M., Engberg, U. (eds.) Intl. Conf. on Foundations of Software Science and Computational Structures (FoSSaCS). LNCS, vol. 2303, pp. 205–222. Springer, Heidelberg (2002)
74. Kobayashi, N.: Types and higher-order recursion schemes for verification of higher-order programs. In: Shao, Z., Pierce, B.C. (eds.) Ann. ACM Symp. on Principles of Programming Languages, pp. 416–428. ACM, New York (2009)
75. Kozen, D.: Results on the propositional mu-calculus. Theor. Comput. Sci. **27**, 333–354 (1983)
76. Kupferman, O.: Automata theory and model checking. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
77. Libkin, L.: Elements of Finite Model Theory. Springer, Heidelberg (2004)
78. Long, D.E., Browne, A., Clarke, E.M., Jha, S., Marrero, W.R.: An improved algorithm for the evaluation of fixpoint expressions. In: Dill, D.L. (ed.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 818, pp. 338–350. Springer, Heidelberg (1994)
79. Maksimova, L.L.: Absence of interpolation and of Beth's property in temporal logics with "the next" operation. Sib. Math. J. **32**(6), 109–113 (1991)
80. Martin, D.: Borel determinacy. Ann. Math. **102**, 363–371 (1975)
81. McIver, A., Morgan, C.: Results on the quantitative μ-calculus qMμ. Trans. Comput. Log. **8**(1) (2007)
82. McMillan, K.L.: Interpolation and model checking. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
83. McNaughton, R.: Infinite games played on finite graphs. Ann. Pure Appl. Log. **65**, 149–184 (1993)
84. Mio, M.: Game semantics for probabilistic mu-calculi. Ph.D. thesis, University of Edinburgh (2012)
85. Moschovakis, Y.: Elementary Induction on Abstract Structures. North-Holland, Amsterdam (1974)
86. Moss, L.S.: Coalgebraic logic. Ann. Pure Appl. Log. **96**, 277–317 (1999). Erratum published Ann. Pure Appl. Log. **99**, 241–259 (1999)
87. Mostowski, A.W.: Regular expressions for infinite trees and a standard form of automata. In: Skowron, A. (ed.) Fifth Symposium on Computation Theory. LNCS, vol. 208, pp. 157–168. Springer, Heidelberg (1984)
88. Mostowski, A.W.: Games with forbidden positions. Tech. Rep. 78, University of Gdansk (1991)
89. Muller, D., Schupp, P.: Alternating automata on infinite trees. Theor. Comput. Sci. **54**, 267–276 (1987)
90. Niwiński, D.: Fixed points vs. infinite generation. In: Ann. Symp. on Logic in Computer Science, pp. 402–409. IEEE, Piscataway (1988)
91. Obdrzálek, J.: Clique-width and parity games. In: Duparc, J., Henzinger, T.A. (eds.) Intl. Workshop Computer Science Logic (CSL). LNCS, vol. 4646, pp. 54–68. Springer, Heidelberg (2007)
92. Ong, C.H.L.: On model-checking trees generated by higher-order recursion schemes. In: Ann. Symp. on Logic in Computer Science, pp. 81–90. IEEE, Piscataway (2006)
93. Parikh, R.: The logic of games and its applications. Ann. Discrete Math. **24**, 111–140 (1985)
94. Park, D.: Finiteness is μ-ineffable. Theor. Comput. Sci. **3**, 173–181 (1976)
95. Piterman, N.: From nondeterministic Büchi and Streett automata to deterministic parity automata. Log. Methods Comput. Sci. **3**(3), 1–21 (2007)
96. Place, T., Segoufin, L.: A decidable characterization of locally testable tree languages. Log. Methods Comput. Sci. **7**(4) (2011)
97. Pratt, V.: A decidable μ-calculus: preliminary report. In: Int. Symp. on Foundations of Computer Science, pp. 421–427. IEEE, Piscataway (1981)
98. Rabin, M.O.: Decidability of second-order theories and automata on infinite trees. Trans. Am. Math. Soc. **141**, 1–23 (1969)

99. Rabin, M.O.: Weakly definable relations and special automata. In: Mathematical Logic and Foundations of Set Theory, pp. 1–23 (1970)
100. Safra, S.: On the complexity of ω-automata. In: Int. Symp. on Foundations of Computer Science. IEEE, Piscataway (1988)
101. Salwicki, A.: Formalized algorithmic languages. Bull. Acad. Pol. Sci., Sér. Sci. Math. Astron. Phys. **18**, 227–232 (1970)
102. Schewe, S.: Solving parity games in big steps. In: Arvind, V., Prasad, S. (eds.) Intl. Conf. on Foundations of Software Technology and Theoretical Computer Science. LNCS, vol. 485, pp. 449–460. Springer, Heidelberg (2007)
103. Schewe, S.: An optimal strategy improvement algorithm for solving parity and payoff games. In: Kaminski, M., Martini, S. (eds.) Intl. Workshop Computer Science Logic (CSL). LNCS, vol. 5213, pp. 369–384. Springer, Heidelberg (2008)
104. Schewe, S.: Büchi complementation made tight. In: Albers, S., Marion, J. (eds.) Annual Symposium on Theoretical Aspects of Computer Science. Leibniz International Proceedings in Informatics, vol. 3, pp. 661–672. Schloss Dagstuhl—Leibniz-Zentrum fuer Informatik, Dagstuhl (2009)
105. Scott, D., de Bakker, J.: A theory of programs. (1969). Unpublished notes, IBM, Vienna (1969)
106. Seidl, H.: Deciding equivalence of finite tree automata. SIAM J. Comput. **19**, 424–437 (1990)
107. Seidl, H.: Fast and simple nested fixpoints. Inf. Process. Lett. **59**(6), 303–308 (1996)
108. Seidl, H., Neumann, A.: On guarding nested fixpoints. In: Flum, J., Rodríguez-Artalejo, M. (eds.) Intl. Workshop Computer Science Logic (CSL). LNCS, vol. 1683, pp. 484–498. Springer, Heidelberg (1999)
109. Semenov, A.: Decidability of monadic theories. In: Chytil, M., Koubek, V. (eds.) International Symposium on Mathematical Foundations of Computer Science. LNCS, vol. 176, pp. 162–175. Springer, Heidelberg (1984)
110. Streett, R.S.: Propositional dynamic logic of looping and converse is elementarily decidable. Inf. Control **54**, 121–141 (1982)
111. Streett, R.S., Emerson, E.A.: The propositional mu-calculus is elementary. In: Paredaens, J. (ed.) International Colloquium on Automata, Languages and Programming. LNCS, vol. 172, pp. 465–472. Springer, Heidelberg (1984)
112. Streett, R.S., Emerson, E.A.: An automata theoretic procedure for the propositional mu-calculus. Inf. Comput. **81**, 249–264 (1989)
113. Thomas, W.: Languages, automata, and logic. In: Rozenberg, G., Salomaa, A. (eds.) Handbook of Formal Languages, vol. III, pp. 389–455. Springer, Heidelberg (1997)
114. Thomas, W.: Infinite games and verification. In: Brinksma, E., Larsen, K.G. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 2404, pp. 58–64. Springer, Heidelberg (2002)
115. Thomas, W.: Constructing infinite graphs with a decidable MSO-theory. In: Rovan, B., Vojtás, P. (eds.) International Symposium on Mathematical Foundations of Computer Science. LNCS, vol. 2747, pp. 113–124. Springer, Heidelberg (2003)
116. Thomas, W.: Church's problem and a tour through automata theory. In: Avron, A., Dershowitz, N., Rabinovich, A. (eds.) Pillars of Computer Science, Lecture Notes in Computer Science, vol. 4800, pp. 635–655. Springer, Heidelberg (2008)
117. van Benthem, J.: Modal Logic and Classical Logic. Bibliopolis, Napoli (1983)
118. Vardi, M.: Reasoning about the past with two-way automata. In: Larsen, K.G., Skyum, S., Winskel, G. (eds.) International Colloquium on Automata, Languages and Programming. LNCS, vol. 1443, pp. 628–641. Springer, Heidelberg (1998)
119. Vardi, M.Y.: The Büchi complementation saga. In: Thomas, W., Weil, P. (eds.) Annual Symposium on Theoretical Aspects of Computer Science. LNCS, vol. 4393, pp. 12–22. Springer, Heidelberg (2007)

120. Vardi, M.Y., Wilke, T.: Automata: from logics to algorithms. In: Flum, J., Grädel, E., Wilke, T. (eds.) Logic and Automata, Texts in Logic and Games, vol. 2, pp. 629–736. Amsterdam University Press, Amsterdam (2007)
121. Venema, Y.: Automata and fixed point logic: a coalgebraic perspective. Inf. Comput. **204**(4), 637–678 (2006)
122. Vöge, J., Jurdziński, M.: A discrete strategy improvement algorithm for solving parity games (extended abstract). In: Emerson, E.A., Sistla, A.P. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 1855, pp. 202–215. Springer, Heidelberg (2000)
123. Walukiewicz, I.: Completeness of Kozen's axiomatisation of the propositional μ-calculus. Inf. Comput. **157**, 142–182 (2000)
124. Walukiewicz, I.: Monadic second order logic on tree-like structures. Theor. Comput. Sci. **257**(1–2), 311–346 (2002)
125. Zielonka, W.: Infinite games on finitely coloured graphs with applications to automata on infinite trees. Theor. Comput. Sci. **200**, 135–183 (1998)

Chapter 27
Graph Games and Reactive Synthesis

Roderick Bloem, Krishnendu Chatterjee, and Barbara Jobstmann

Abstract Graph-based games are an important tool in computer science. They have applications in synthesis, verification, refinement, and far beyond. We review graph-based games with objectives on infinite plays. We give definitions and algorithms to solve the games and to give a winning strategy. The objectives we consider are mostly Boolean, but we also look at quantitative graph-based games and their objectives. Synthesis aims to turn temporal logic specifications into correct reactive systems. We explain the reduction of synthesis to graph-based games (or equivalently tree automata) using synthesis of LTL specifications as an example. We treat the classical approach that uses determinization of parity automata and more modern approaches.

27.1 Introduction

Reactive synthesis is the problem of automatically constructing a correct reactive system from a given specification [75]. Graph games (or, equivalently, tree automata) are a central tool in solving the synthesis problems [30, 101]. In this chapter, we shall review the theory of games and synthesis. Besides reactive synthesis, Syntax-Guided Synthesis (SYGUS) has become a popular and promising approach to automatically generate (parts of) programs from specifications. SYGUS differs from reactive synthesis in its goals and especially in the techniques it uses: it is typically done inductively instead of deductively, and is often driven by counterexamples (hence the related term Counterexample-Guided Inductive Synthesis or CEGIS). We refer the interested reader to [8] and the references contained in that paper.

R. Bloem (✉)
Graz University of Technology, Graz, Austria
e-mail: roderick.bloem@iaik.tugraz.at

K. Chatterjee
IST Austria, Klosterneuburg, Austria

B. Jobstmann
École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland

E.M. Clarke et al. (eds.), *Handbook of Model Checking*,
DOI 10.1007/978-3-319-10575-8_27

We consider two-player perfect-information nonterminating games played on graphs, that proceed for an infinite number of rounds. The state of a game is a vertex of a graph. The graph is partitioned into player-1 states and player-2 states: in player-1 states, player 1 chooses the successor vertex; in player-2 states, player 2 chooses the successor vertex. In each round, the state changes along an edge of the graph to a successor vertex. Thus, the outcome of the game being played for an infinite number of rounds is an infinite path through the graph. These games play a central role in several areas of computer science. One important application arises when the vertices and edges of a graph represent the states and transitions of a reactive system, and the two players represent controllable versus uncontrollable decisions during the execution of the system, which corresponds to the *synthesis* problem for reactive systems. Game-theoretic formulations have proved useful not only for synthesis, but also for the modeling [1, 79], refinement [104], verification [6, 9], testing [17], and compatibility checking [3, 4] of reactive systems. The use of ω-regular objectives is natural in these application contexts. This is because the winning conditions of the games arise from requirements specifications for reactive systems, and the ω-regular sets of infinite paths provide an important and robust paradigm for such specifications [124].

Synthesis is the problem of automatically constructing a correct reactive system from a given specification. Thus, synthesis goes beyond verification, in which both a specification and an implementation have to be given, by automatically deriving the latter from the former. Synthesis is thus a fundamental approach that aims at moving the construction of reactive systems from the imperative to the declarative level. If one is convinced that a complete specification should be written before the implementation is constructed, synthesis is a natural and important endeavor.

In this chapter, we employ the term synthesis exclusively in the setting of synchronous reactive systems, which maintain a constant interaction with the environment. We will assume that both the inputs and the outputs of such a system are Boolean. One way to specify the behavior of such systems is as a set of infinite words over the input and output valuations. The distinction between inputs and outputs is crucial: outputs are under direct control of the system whereas inputs are not. Thus, at any point in time, we must find some output that works for all inputs. More precisely, given a finite input sequence, we must find some output that allows a correct execution for any possible future input.

The synthesis question and the corresponding decidability problem, called *realizability*, were originally posed by Church [75], who used the monadic second-order logic of one successor (S1S) as a specification language. The problem was solved by Rabin [146] and by Büchi and Landweber [30] in the late 1960s. In this chapter, we will focus on the more modern Linear Temporal Logic (LTL) [141]. Initial work on synthesizing systems from temporal specifications assumed cooperative environments and reduced the synthesis problem to satisfiability [76, 84, 126]. The problem of synthesizing systems from specifications in LTL (in adversary environments, which are the focus of this chapter) was studied in [142], where it was shown that the problem is complete for 2EXPTIME. Even though the complexity of the synthesis problem from LTL is significantly lower than that from S1S, which

is non-elementary [166], it discouraged researchers and led to few developments for several decades. This relative silence has been followed by a flurry of activity since around 2005. In this chapter, we will give an overview of both the classical approach to LTL synthesis and the relatively practical approaches that have been proposed recently.

The question of synthesis can be generalized to *controller synthesis*: the question of finishing an incompletely specified system. This problem was first studied by Ramadge and Wonham [148] under the name of Synthesis of Discrete Event Systems. The question here is to control a *plant* in such a way that it fulfills its specification. Again, we must distinguish between the inputs of the plant, which are determined by an uncontrollable environment, and its control parameter: the control parameter must be adjusted continually in such a way that the plant works correctly for any input. The original approach aimed at safety properties only, but can be extended to more expressive specification formalisms using the same techniques that are used in synthesis. Note that in synthesis we have only a specification, whereas in controller synthesis we have both a specification and an incomplete implementation. The distinction is somewhat fluid as implementations can be expressed in temporal logic (using extra variables), while specifications can be expressed as automata, which are a form of transition systems.

A standard approach to verification is automata theoretic: we build an automaton corresponding to the negation of the specification and construct the product with the system that we wish to verify. The system can evolve in several ways, depending on the inputs, which means that the product is nondeterministic, and multiple paths must be searched for an incorrect execution. In synthesis, in contrast, we have two types of freedom: the freedom to choose the inputs, which we cannot control, and the freedom to choose the outputs, which are under our control. Thus, instead of a nondeterministic automaton, the natural model here is a game, or more precisely, an infinite zero-sum graph-based game with two players, which is won iff the specification is satisfied. (Equivalently, we can use tree automata.)

In the next section, we will discuss game theory, with a focus on the games that arise in synthesis settings. We will start with qualitative games, i.e., we will look at games with various Boolean winning conditions that occur in practice. Then, we will consider quantitative games, which occur when we consider more subtle specifications. In Sect. 27.3 we will discuss the classical and some more modern algorithms for LTL synthesis. In Sect. 27.4, we will conclude with related work that we cannot discuss in full detail.

27.2 Theory of Graph-Based Games

In this section we present definitions of game graphs, plays, strategies, and objectives. We will define when a game is won and introduce the appropriate decision problems. We will then discuss the basic techniques and algorithms to solve them.

27.2.1 Game Graphs and Strategies

Game Graphs. A *game graph* $G = \langle (S, E), (S_1, S_2) \rangle$ consists of a *finite* set S of states partitioned into player-1 states S_1 and player-2 states S_2 (i.e., $S = S_1 \cup S_2$ and $S_1 \cap S_2 = \emptyset$), and a set $E \subseteq S \times S$ of edges such that for all $s \in S$, there exists (at least one) $t \in S$ such that $(s, t) \in E$. In other words, every state has at least one outgoing edge. A *player*-1 *game* is a game graph where $S_1 = S$ and $S_2 = \emptyset$, and vice versa for player 2. The sub-graph of G induced by $U \subseteq S$ is the graph $\langle (U, E \cap (U \times U)), (U \cap S_1, U \cap S_2) \rangle$ (which is not a game graph in general); the sub-graph induced by U is a game graph if for all $s \in U$ there exists a $t \in U$ such that $(s, t) \in E$.

Plays and Strategies. A game on G starting from a state $s_0 \in S$ is played in rounds as follows. If the game is in a player-1 state, then player 1 chooses an outgoing edge to determine the successor state; otherwise the game is in a player-2 state, and player 2 chooses the successor state. This way, the game results in a *play* from s_0, i.e., an infinite path $\rho = s_0 s_1 \cdots \in S^\omega$ such that $(s_i, s_{i+1}) \in E$ for all $i \geq 0$. We denote the set of all plays as $\mathsf{Plays}(G)$. The prefix of length n of ρ is denoted by $\rho(n)$. We often identify ρ with the set of states in ρ, and we use expressions such as $s_0 \in \rho$. A *strategy* for player 1 is a recipe that prescribes how to extend the prefix of a play. Formally, a strategy σ for player 1 is a function $\sigma : S^* S_1 \to S$ such that $(s, \sigma(w \cdot s)) \in E$ for all $w \in S^*$ and $s \in S_1$. An *outcome* of σ from s_0 is a play $s_0 s_1 \ldots$ such that $\sigma(s_0 \ldots s_i) = s_{i+1}$ for all $s_i \in S_1$. Strategy and outcome for player 2 are defined analogously. We denote by Σ and Π the set of strategies for player 1 and player 2, respectively. Given strategies σ and π for player 1 and 2, respectively, and a starting state s_0, there is a unique play (or outcome) $s_0 s_1 \ldots$, denoted $\rho(s_0, \sigma, \pi)$, such that for all $i \geq 0$, if $s_i \in S_1$, then $s_{i+1} = \sigma(s_0 s_1 \ldots s_i)$ and if $s_i \in S_2$, then $s_{i+1} = \pi(s_0 s_1 \ldots s_i)$.

Finite-Memory and Memoryless Strategies. A strategy uses *finite-memory* if it can be encoded by a deterministic transducer $\langle M, m_0, \sigma_u, \sigma_n \rangle$ where M is a finite set (the memory of the strategy), $m_0 \in M$ is the initial memory value, $\sigma_u : M \times S \to M$ is an update function, and $\sigma_n : M \times S_1 \to S$ is a next-move function. The *size* of the strategy is the number $|M|$ of memory values. If the game is in a player-1 state s, the strategy chooses $t = \sigma_n(m, s)$ as the next state (where m is the current memory value), and the memory is updated to $\sigma_u(m, s)$. Formally, $\langle M, m_0, \sigma_u, \sigma_n \rangle$ defines the strategy σ such that $\sigma(w \cdot s) = \sigma_n(\hat{\sigma}_u(m_0, w), s)$ for all $w \in S^*$ and $s \in S_1$, where $\hat{\sigma}_u$ extends σ_u to sequences of states in the usual way. A strategy is *memoryless* if it is independent of the history of the play and depends only on the current state. In other words, a memoryless strategy σ has only one memory state, i.e., $|M| = 1$, and hence the strategy is specified as $\sigma : S_1 \to S$. For a finite-memory strategy σ, G_σ is the graph obtained as the product of G with the transducer defining σ, where $(\langle m, s \rangle, \langle m', s' \rangle)$ is a transition in G_σ if $m' = \sigma_u(m, s)$ and either $s \in S_1$ and $s' = \sigma_n(m, s)$, or $s \in S_2$ and $(s, s') \in E$.

27.2.2 Objectives

In this section we will define objectives. An *objective* for G is a set $\varphi \subseteq S^\omega$.

Qualitative Objectives. For an infinite play ρ we denote by $\mathsf{Inf}(\rho)$ the set of states that occur infinitely often in ρ. We consider the following objectives:

- *Reachability Objectives.* A reachability objective is defined by a set $F \subseteq S$ of target states, and the objective requires that a state in F is visited at least once. Formally, $\mathsf{Reach}_G(F) = \{\rho \in \mathsf{Plays}(G) \mid \exists s \in \rho : s \in F\}$. The dual of reachability objectives are safety objectives, and a safety objective is defined by a set $F \subseteq S$ of safe states, and the objective requires that only states in F are visited. Formally, $\mathsf{Safe}_G(F) = \{\rho \in \mathsf{Plays}(G) \mid \forall s \in \rho : s \in F\}$.
- *Büchi Objectives.* A Büchi objective is defined by a set $B \subseteq S$ of target states, and the objective requires that a state in B is visited infinitely often. Formally, $\mathsf{Buchi}_G(B) = \{\rho \in \mathsf{Plays}(G) \mid \mathsf{Inf}(\rho) \cap B \neq \emptyset\}$. Büchi objectives represent liveness specifications, and the dual of a Büchi objective is called a co-Büchi objective. A co-Büchi objective consists of a set $C \subseteq S$ of states and requires states outside C to be visited finitely often, i.e., $\mathsf{coBuchi}_G(C) = \{\rho \in \mathsf{Plays}(G) \mid \mathsf{Inf}(\rho) \subseteq C\}$.
- *Rabin and Streett Objectives.* Rabin and Streett objectives are obtained as Boolean combinations of Büchi and co-Büchi objectives. A *Rabin specification* for the game graph G is a finite set $R = \{(E_1, F_1), \ldots, (E_d, F_d)\}$ of pairs of sets of states, that is, $E_j \subseteq S$ and $F_j \subseteq S$ for all $1 \leq j \leq d$. The pairs in R are called Rabin pairs. We assume without loss of generality that $\bigcup_{1 \leq j \leq d}(E_j \cup F_j) = S$. The Rabin objective requires that for some $1 \leq j \leq d$, all states in the left-hand set E_j are visited finitely often, and some state in the right-hand set F_j is visited infinitely often. Thus, the Rabin objective defined by R is the set $\mathsf{Rabin}_G(R) = \{\rho \in \mathsf{Plays}(G) \mid (\exists 1 \leq j \leq d)(\mathsf{Inf}(\rho) \cap E_j = \emptyset \wedge \mathsf{Inf}(\rho) \cap F_j \neq \emptyset)\}$ of winning paths. Note that the co-Büchi objective $\mathsf{coBuchi}_G(C)$ is equal to the single-pair Rabin objective $\mathsf{Rabin}_G(\{(S \setminus C, S)\})$, and the Büchi objective $\mathsf{Buchi}_G(B)$ is equal to the two-pair Rabin objective $\mathsf{Rabin}_G(\{(\emptyset, B), (S, S)\})$.[1] The complements of Rabin objectives are called Streett objectives. A *Streett specification* for G is likewise a set $Q = \{(E_1, F_1), \ldots, (E_d, F_d)\}$ of pairs of sets of states $E_j \subseteq S$ and $F_j \subseteq S$ such that $\bigcup_{1 \leq j \leq d}(E_j \cup F_j) = S$. The pairs in Q are called Streett pairs. The Streett objective Q requires that for every Streett pair (E_j, F_j), $1 \leq j \leq d$, if some state in the right-hand set F_j is visited infinitely often, then some state in the left-hand set E_j is visited infinitely often. Formally, the Streett objective defined by Q is the set $\mathsf{Streett}_G(Q) = \{\rho \in \mathsf{Plays}(G) \mid (\forall 1 \leq j \leq d)(\mathsf{Inf}(\rho) \cap E_j \neq \emptyset \vee \mathsf{Inf}(\rho) \cap F_j = \emptyset)\}$ of winning paths. Note that $\mathsf{Streett}_G(Q) = \mathsf{Plays}(G) \setminus \mathsf{Rabin}_G(Q)$.

[1] Note that no run can satisfy the condition expressed by the second pair, which is however required by the definition.

- *Parity Objectives.* Let $p : S \to \mathbb{N}_0$ be a *priority function*. The *parity* objective $\mathsf{Parity}_G(p) = \{\rho \in \mathsf{Plays}(G) \mid \min\{p(s) \mid s \in \mathsf{Inf}(\rho)\} \text{ is even}\}$ requires that the minimum of the priorities of the states visited infinitely often be even. The special cases of *Büchi* and *co-Büchi* objectives correspond to the case with two priorities, $p : S \to \{0, 1\}$ and $p : S \to \{1, 2\}$ respectively. (Here, priorities 0 and 2 are for the accepting states, 1 is for the rejecting states.)

We refer to the above objectives as qualitative objectives since they are defined by Boolean combinations of sets that are subsets of S.

Relationship Between Rabin, Streett and Parity Objectives. We have already seen how Büchi and co-Büchi objectives are special cases of Rabin, Streett and parity objectives. We now present the relationship between Rabin, Streett and parity objectives. Parity objectives are also called *Rabin-chain* objectives, as they are a special case of Rabin objectives [169]: if the sets of a Rabin specification $R = \{(E_1, F_1), \ldots, (E_d, F_d)\}$ form a chain $E_1 \subsetneq F_1 \subsetneq E_2 \subsetneq F_2 \subsetneq \cdots \subsetneq E_d \subsetneq F_d$, then $\mathsf{Rabin}_G(R) = \mathsf{Parity}_G(p)$ for the priority function $p\colon S \to \{0, 1, \ldots, 2d\}$ that for every $1 \leq j \leq d$ assigns to each state in $E_j \setminus F_{j-1}$ the priority $2j - 1$, and to each state in $F_j \setminus E_j$ the priority $2j$, where $F_0 = \emptyset$. Conversely, given a priority function $p\colon S \to \{0, 1, \ldots, 2d\}$, we can construct a chain $E_1 \subsetneq F_1 \subsetneq \cdots \subsetneq E_{d+1} \subsetneq F_{d+1}$ of $d + 1$ Rabin pairs such that $\mathsf{Parity}_G(p) = \mathsf{Rabin}_G(\{(E_1, F_1), \ldots, (E_{d+1}, F_{d+1})\})$ as follows: let $E_1 = \emptyset$ and $F_1 = p^{-1}(0)$, and for all $1 \leq j \leq d + 1$, let $E_j = F_{j-1} \cup p^{-1}(2j - 3)$ and $F_j = E_j \cup p^{-1}(2j - 2)$. Hence, the parity objectives are a subclass of the Rabin objectives that is closed under complementation. It follows that every parity objective is both a Rabin objective and a Streett objective. The parity objectives are of special interest, because every ω-regular objective can be turned into a parity objective by modifying the game graph (take the synchronous product of the game graph with a deterministic parity automaton that accepts the ω-regular objective) [133]. Moreover, parity objectives enjoy several attractive computational properties (see discussion of algorithms for parity games in Sect. 27.2.4).

Quantitative Objectives. We consider three classical quantitative objectives defined with weight functions on the edges of the graph. Let $w : E \to \mathbb{Z}$ be a *weight function*, where positive numbers represent rewards. We denote by W the largest weight (in absolute value) according to w.

- *Energy Objectives.* Given a play ρ, the *energy level* of a prefix $\gamma = s_0 s_1 \ldots s_n$ of the play is $\mathsf{EL}(\gamma) = \sum_{i=0}^{n-1} w((s_i, s_{i+1}))$. Given an initial credit $c_0 \in \mathbb{N} \cup \{\infty\}$, the *energy* objective $\mathsf{PosEnergy}_G(c_0) = \{\rho \in \mathsf{Plays}(G) \mid \forall n \geq 0 : c_0 + \mathsf{EL}(\rho(n)) \geq 0\}$ requires that the energy level is always non-negative.
- *Mean-Payoff Objectives.* The *mean-payoff value* of a play $\rho = s_0 s_1 \ldots$ is $\mathsf{MP}(\rho) = \liminf_{n\to\infty} \frac{1}{n} \cdot \mathsf{EL}(\rho(n))$. Given a threshold $\theta \in \mathbb{Q}$, the *mean-payoff* objective $\mathsf{MeanPayoff}_G(\theta) = \{\rho \in \mathsf{Plays}(G) \mid \mathsf{MP}(\rho) \geq \theta\}$ requires that the mean-payoff value be at least θ.
- *Discounted Objectives.* Given a discount factor $0 < \lambda < 1$, the *discounted value* of a play $\rho = s_0 s_1 \ldots$ is $\mathsf{Disc}(\lambda, \rho) = \sum_{i=0}^{\infty} \lambda^i \cdot w((s_i, s_{i+1}))$. Given a

threshold $\theta \in \mathbb{Q}$, the *discounted* objective $\mathsf{Discounted}_G(\lambda, \theta) = \{\rho \in \mathsf{Plays}(G) \mid \mathsf{Disc}(\lambda, \rho) \geq \theta\}$ requires that the discounted value be at least θ.

In the sequel, when the game G is clear from the context, we omit the subscript in objective names.

27.2.3 Winning and Optimal Strategies; Decision Problems

We now define the notion of winning in games and decision problems.

Winning Strategies and Sets. Given a game graph G, a starting state s_0 and an objective φ, a strategy σ is *winning* for player 1 from s_0 for φ if for all strategies π for player 2 we have $\rho(s_0, \sigma, \pi) \in \varphi$. The set of winning states $W_1(\varphi) = \{s_0 \mid \exists \sigma \in \Sigma.\ \forall \pi \in \Pi.\ \rho(s_0, \sigma, \pi) \in \varphi\}$ is the set of states s_0 such that player 1 has a winning strategy from s_0 for φ (note that an objective is a set of plays). The winning set $W_2(\varphi) = \{s_0 \mid \exists \pi \in \Pi.\ \forall \sigma \in \Sigma.\ \rho(s_0, \sigma, \pi) \in \varphi\}$ is defined analogously. We will consider the winning sets and strategies for objectives defined in the previous subsection, i.e., reachability, safety, Büchi, co-Büchi, parity, Rabin, Streett, energy, mean-payoff and discounted objectives. For energy objectives, we will also consider the *finite initial credit* problem, where the winning region is the set of states s_0 such that there exists a finite initial credit c_0 such that $s_0 \in W_1(\mathsf{PosEnergy}_G(c_0))$.

Decision Problems. The decision problems that we consider consist of an input game graph G, an objective φ and a state s_0, and the decision problem asks whether $s_0 \in W_1(\varphi)$. We will also consider the decision problem for the finite initial credit problem, which asks whether a given state s_0 is in the winning set for the finite initial credit problem.

27.2.4 Complexity and Algorithms for Graph Games with Qualitative Objectives

In this section we will discuss the results related to graph games with qualitative objectives. We will focus on the strategy complexity, computational complexity, and algorithms. We will mention the basic techniques and relevant pointers to the literature. We will first discuss symbolic algorithms for game solving.

Symbolic Algorithms. The symbolic algorithms for game solving are obtained by characterizing the winning set using μ-calculus formulae (cf. Chap. 26 in this Handbook [24]). A μ-calculus formula is a succinct description of a nested iterative algorithm that uses only set operations and the predecessor operators (described in the next paragraph). All the set operations and predecessor computations are symbolic

steps that are available as primitive operations in, e.g., a BDD library (cf. Chap. 7 in this Handbook [29]) such as CuDD [164]. Thus, a μ-calculus formula for the winning set presents a symbolic algorithm for game solving. We will describe the μ-calculus formula for reachability and Büchi games. The μ-calculus formulas for parity games are presented in [86] and they were later generalized to Rabin and Streett games in [138].

Reachability and Safety Games. We first present the classical algorithm to solve reachability games. Let us first define the *predecessor* operator. Given a set $X \subseteq S$ of states, the predecessor operator $\mathsf{Pre}_1(X)$ is defined as follows

$$\mathsf{Pre}_1(X) = \big\{s \in S_1 \bigm| \exists t \in X.(s,t) \in E\big\} \cup \big\{s \in S_2 \bigm| \forall t \in S.(s,t) \in E \rightarrow t \in X\big\}.$$

In other words, $\mathsf{Pre}_1(X)$ is the set of states such that either the state is a player-1 state and there is a next state in X or the state is a player-2 state and all choices lead to a next state in X. The dual predecessor operator is as follows:

$$\mathsf{Pre}_2(X) = \big\{s \in S_2 \bigm| \exists t \in X.(s,t) \in E\big\} \cup \big\{s \in S_1 \bigm| \forall t \in S.(s,t) \in E \rightarrow t \in X\big\}.$$

The classical algorithm for games with reachability objectives is the fixpoint computation of the $\mathsf{Pre}_1(\cdot)$ operator. Given a target set T, let $T_0 = T$, and for $i \geq 0$, let T_{i+1} be defined inductively as $T_{i+1} := T_i \cup \mathsf{Pre}_1(T_i)$. Let us consider the fixpoint, which is also called the *attractor* of player 1 to T. Let $T_* = \mathsf{Attr}_1(T) = \bigcup_{i \geq 0} T_i$. A memoryless strategy for player 1 for T_* is defined as follows: for a state $s \in (T_{i+1} \setminus T_i) \cap S_1$ choose an edge (s,t) such that $t \in T_i$ (such an edge exists by construction). It is easy to show by induction that for all $s \in T_i$, player 1 can ensure to reach T within i steps against all player-2 strategies. Hence $T_* \subseteq W_1(\mathsf{Reach}(T))$. Let $\overline{T}_* = S \setminus T_*$. For all $s \in \overline{T}_* \cap S_1$, there are no outgoing edges (s,t) with $t \in T_*$ (otherwise, s would have been included in T_*); and for all $s \in \overline{T}_* \cap S_2$, there is an outgoing edge (s,t) with $t \in \overline{T}_*$ (otherwise, s would have been included in T_*). A memoryless strategy for player 2 that for all $s \in \overline{T}_* \cap S_2$ chooses an outgoing edge (s,t) with $t \in \overline{T}_*$ ensures against all player-1 strategies that $\overline{T}_*$ is not left. Thus we have $\overline{T}_* \subseteq W_2(\mathsf{Safe}(S \setminus T))$. A linear-time algorithm to compute $W_1(\mathsf{Reach}(T))$ is given in [13, 106]. We summarize the main results of reachability and safety games in the following theorem.

Theorem 1 *Given a game graph G with n vertices and m edges and a target set T, the following assertions hold*:

1. $W_1(\mathsf{Reach}(T)) = S \setminus W_2(\mathsf{Safe}(S \setminus T))$ *and memoryless winning strategies exist for both players.*
2. *The winning set $W_1(\mathsf{Reach}(T))$ can be computed in $O(m)$ (linear) time.*
3. *The winning set $W_1(\mathsf{Reach}(T))$ can be computed symbolically with the μ-calculus formula $\mu X.[T \cup \mathsf{Pre}_1(X)]$.*

Büchi and co-Büchi Games. The algorithm for Büchi games is obtained by repeatedly applying the attractor computation (reachability game solutions). Informally,

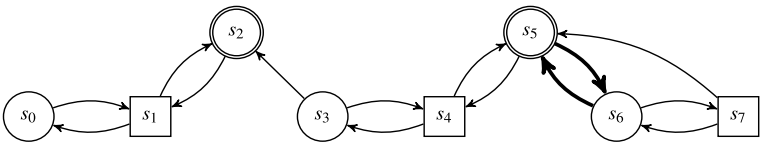


Fig. 1 Büchi game of Example 1

the algorithm is as follows: let B be the set of Büchi states. We first compute the set $A_1 = W_1(\mathsf{Reach}(B)) = \mathsf{Attr}_1(B)$ such that player 1 has a strategy to reach B at least once. In the complement set of A_1, player 1 cannot even reach B once and hence is clearly not winning. The complement set $\overline{A}_1$ and the player-2 attractor $\mathsf{Attr}_2(\overline{A}_1)$ are removed from the graph. The process is iterated unless the set $\overline{A}_1$ is empty. If $\overline{A}_1$ is empty, then from all states in A_1 player 1 can ensure to reach B and stay in A_1 and hence ensure that the set B is visited infinitely often. We now formally describe an iteration j of the algorithm: the set of states at iteration j is denoted by S^j, the game graph by G^j, and the set of Büchi states $B \cap S^j$ by B^j. Given a game graph G and a set U of states, we denote by $G \restriction U$ the game graph induced by U. We have that G^j is the game graph induced by S^j. At iteration j, the algorithm first finds the set of states A_1^j from which player 1 can ensure that the play reaches the set B^j, i.e., computes $\mathsf{Attr}_1(B^j)$ in G^j. The rest of the states $\overline{A}_1^j = S^j \setminus A_1^j$ are winning for player 2. Then the set of states W_{j+1}, from which player 2 can ensure reaching $\overline{A}_1^j$ i.e., $\mathsf{Attr}_2(\overline{A}_1^j)$ in G^j, is computed. The set W_{j+1} is winning for player 2, and *not* for player 1 in G^j and also in G. Thus, it is removed from the vertex set to obtain game graph G^{j+1}. The algorithm then iterates on the reduced game graph, i.e., proceeds to iteration $j+1$ on G^{j+1}. The correctness proof of the algorithm shows that when the algorithm terminates, all the remaining states are winning for player 1. The pseudocode of the algorithm is described in Algorithm 1. For improved algorithms for Büchi games see [52, 53, 61, 64].

Example 1 We illustrate the algorithm for Büchi games on the example game graph shown in Fig. 1. The player-1 states are depicted as circles and player-2 states as boxes; and Büchi states are indicated as double circles. The set $\overline{A}_1^0$ is the set $\{s_0, s_1\}$ and its player-2 attractor is $\{s_0, s_1, s_2\}$. In the following iteration the set $\overline{A}_1^1$ is the set $\{s_3, s_4\}$ and its player-2 attractor is also the set $\{s_3, s_4\}$. In the next iteration the set $\overline{A}_1^2$ is empty, and thus the algorithm returns $(\{s_5, s_6, s_7\}, \{s_0, s_1, s_2, s_3, s_4\})$. The winning strategy for player 1 in the set $\{s_5, s_6, s_7\}$ (indicated by bold arrows in Fig. 1) is as follows: in state s_5 choose the successor s_6 and in s_6 choose the successor s_5.

Characterization of Winning Set by μ-calculus Formula. The μ-calculus formula to characterize the winning set for Büchi objectives is as follows:

$$\nu Y.\mu X.\big[\big(B \cap \mathsf{Pre}_1(Y)\big) \cup \big((S \setminus B) \cap \mathsf{Pre}_1(X)\big)\big].$$

Algorithm 1: Classical algorithm for Büchi Games

Input : A game graph $G = \langle (S, E), (S_1, S_2) \rangle$ and $B \subseteq S$.
Output: $(S \setminus W, W)$: the winning set partition.
1. $G^0 := G$; $S^0 := S$; 2. $W_0 := \emptyset$; 3. $j := 0$
4. **repeat**
   4.1 $W_{j+1} := \mathsf{AvoidSetClassical}(G^j, B \cap S^j)$
   4.2 $S^{j+1} := S^j \setminus W_{j+1}$; $G^{j+1} = G \restriction S^{j+1}$; $j := j + 1$;
   until $W_j = \emptyset$
5. $W := \bigcup_{k=1}^{j} W_k$;
6. **return** $(S \setminus W, W)$.

Procedure AvoidSetClassical
Input: Game graph G^j and $B^j \subseteq S^j$.
Output: set $W_{j+1} \subseteq S^j$.
1. $A_1^j := \mathsf{Attr}_1(B^j)$ in G^j; 2. $\overline{A}_1^j := S^j \setminus A_1^j$; 3. $W_{j+1} := \mathsf{Attr}_2(\overline{A}_1^j)$ in G^j;

The argument that the above formula gives the winning set for Büchi objectives is as follows: let $Y_* = \nu Y.\mu X.[(B \cap \mathsf{Pre}_1(Y)) \cup ((S \setminus B) \cap \mathsf{Pre}_1(X))]$. Since Y_* is a fixpoint, if we replace Y by Y_* we would obtain the same result, i.e.,

$$Y_* = \mu X.\big[\big(B \cap \mathsf{Pre}_1(Y_*)\big) \cup \big((S \setminus B) \cap \mathsf{Pre}_1(X)\big)\big].$$

Let $T = B \cap Y_*$; and all states in $B \cap Y_*$ satisfy $\mathsf{Pre}_1(Y_*)$, i.e., in states of $B \cap Y_*$ player 1 can ensure that the next state is in Y_*. Treating the set $T = B \cap Y_* = B \cap \mathsf{Pre}_1(Y_*)$ as the target set for reachability objectives, it follows that for all states in Y_* player 1 can ensure to reach T. Hence player 1 can ensure to reach T from all states in Y_* and Y_* is never left, and thus T is visited infinitely often. Since $T \subseteq B$, it follows that the Büchi objective is satisfied. A similar argument shows that in the complement of the μ-calculus formula, player 2 can ensure the complement co-Büchi objective (we also refer the reader to Chap. 26 in this Handbook [24] for an excellent exposition on μ-calculus). The main results for Büchi games are summarized as follows.

Theorem 2 *Given a game graph G, with set B of Büchi states, the following assertions hold*:

1. $W_1(\mathsf{Buchi}(B)) = S \setminus W_2(\mathsf{coBuchi}(S \setminus B))$ *and memoryless winning strategies exist for both players.*
2. *The winning set $W_1(\mathsf{Buchi}(B))$ can be computed in $O(n \cdot m)$ (quadratic) time.*
3. *The winning set $W_1(\mathsf{Buchi}(B))$ can be computed symbolically with the μ-calculus formula $\nu Y.\mu X.[(B \cap \mathsf{Pre}_1(Y)) \cup ((S \setminus B) \cap \mathsf{Pre}_1(X))]$.*

Parity Games. Emerson and Jutla [86] established the equivalence of solving 2-player parity games and μ-calculus model checking (see Chap. 26 in this Hand-

book [24]). This intriguing connection led to much research attempting to solve 2-player parity games in polynomial time. Alas, the problem is still open. The classical algorithm for solving parity games proceeds by a recursive decomposition of the problem and repeatedly solving games with reachability objectives [128, 169]. The algorithm generalizes the algorithm presented for Büchi games, and the correctness proof establishes the existence of memoryless winning strategies for both players. The running time of the algorithm for games with n states, m edges, and d priorities is $O(n^{d-1} \cdot m)$. Jurdziński [111] gave an improved algorithm to solve parity games based on a notion of ranking functions and progress measures. This algorithm, called the *small progress measure* algorithm, has a running time of $O((\frac{2n}{d})^{\lfloor \frac{d}{2} \rfloor} \cdot m)$; moreover, there exists a family of games on which the running time of the algorithm is exponential. Another notable algorithm for solving parity games is the *strategy improvement* algorithm [175]. This algorithm iterates local optimizations of memoryless strategies which converge to a globally optimal strategy. Also see [154] for another strategy improvement scheme. Based on the strategy improvement algorithm, a randomized subexponential-time algorithm (with an expected running time of $O(2^{\sqrt{n \cdot \log n}})$) for solving parity games was presented by Björklund et al. [15]. Friedmann [97] showed that there exists a family of games on which the running time of the strategy-improvement algorithms is exponential, and for a more elaborate description of lower bounds for strategy-improvement schemes see [98]. Jurdziński et al. [112] gave a deterministic subexponential-time algorithm for solving 2-player games with parity objectives. By combining the small progress measure algorithm [111] and the deterministic subexponential-time algorithm [112], an improved algorithm was presented in [153] with roughly $O((\frac{3n}{d})^{\lfloor \frac{d}{3} \rfloor} \cdot m)$ running time. We summarize the results in the following theorem.

Theorem 3 *Given a game graph G, with a priority function p with d priorities, the following assertions hold*:

1. $W_1(\mathsf{Parity}(p)) = S \setminus W_2(\mathsf{Plays}(G) \setminus \mathsf{Parity}(p))$ *and memoryless winning strategies exist for both players.*
2. *Given a state s, whether $s \in W_1(\mathsf{Parity}(p))$ can be decided in NP $\cap$ coNP.*
3. *The winning set $W_1(\mathsf{Parity}(p))$ can be computed in $O((\frac{3n}{d})^{\lfloor \frac{d}{3} \rfloor} \cdot m)$, and also in $n^{O(\sqrt{n})}$ time.*
4. *The winning set $W_1(\mathsf{Parity}(p))$ can be computed symbolically with a μ-calculus formula of alternation depth $d - 1$.*

Rabin and Streett Games. Gurevich and Harrington [101] showed that for 2-player games with ω-regular objectives, finite-memory strategies suffice for winning. The construction of finite-memory winning strategies is based on a data structure, called a *latest appearance record* (LAR), which remembers the order of the latest appearances of the states in a play. Emerson and Jutla [85] established that for 2-player games with Rabin objectives, memoryless strategies suffice for winning. The results of Dziembowski et al. [80] give precise memory requirements for strategies in 2-player games with ω-regular objectives: their construction of strategies is based on

a tree representation of a Muller objective, called the *Zielonka tree*, which was introduced in [180] (for details of Muller objectives see [169, 180]). It follows from these results that for Streett objectives with d-pairs, $d!$ memory is both necessary and sufficient. Emerson and Jutla [85] showed that the solution problem for Rabin objectives is NP-complete, and dually, coNP-complete for Streett objectives. The notable algorithms for games with Rabin and Streett objectives include the adaptation of the classical algorithm of Zielonka [180] for Muller games specialized to Rabin and Streett games (see [105] for an exposition); an algorithm that is based on a reduction to the emptiness problem for weak alternating automata [119]; a generalization of the small progress measure algorithm for parity games to Rabin and Streett games [138]; and a generalization of the subexponential-time algorithm for parity games [112] to Rabin and Streett games [62]. Symbolic algorithms for Rabin and Streett games are presented in [138].

Theorem 4 *Given a game graph G, with a set $P = \{(E_1, F_1), \ldots, (E_d, F_d)\}$ of d pairs of sets of states, the following assertions hold*:

1. *We have $W_1(\mathsf{Rabin}(P)) = S \setminus W_2(\mathsf{Plays}(G) \setminus \mathsf{Rabin}(P))$, and $W_1(\mathsf{Streett}(P)) = S \setminus W_2(\mathsf{Plays}(G) \setminus \mathsf{Streett}(P))$. Memoryless winning strategies exist for Rabin objectives, and for Streett objectives $d!$ memory is necessary and sufficient.*
2. *Given a state s, the decision problem whether $s \in W_1(\mathsf{Rabin}(P))$ is NP-complete, and the decision problem whether $s \in W_1(\mathsf{Streett}(P))$ is coNP-complete.*
3. *The winning sets $W_1(\mathsf{Rabin}(P))$ and $W_1(\mathsf{Streett}(P))$ can be computed in $O(d! \cdot n^d \cdot m)$ time.*

Boolean Combinations. We now discuss some Boolean combinations of the above objectives that have been used in synthesis. The class of GR(1) (Generalized Reactivity(1)) conditions was introduced in [140]. A GR(1) objective is specified as an implication between a conjunction of k_1 Büchi objectives (the assumptions) and a conjunction of k_2 Büchi objectives (the guarantees). A large class of objectives in synthesis can be specified as GR(1) specifications [140], and games with GR(1) conditions can be solved in time $O(n^2 \cdot m \cdot k_1 \cdot k_2)$ [140] and also in time $O(n \cdot m \cdot (k_1 \cdot k_2)^2)$ [18]. Games with generalized parity objectives (conjunction and disjunction of parity objectives) have been studied in [62].

27.2.5 Complexity and Algorithms for Graph Games with Quantitative Objectives

In this section we will discuss the results related to solving graph games with quantitative objectives. Again we will focus on the strategy complexity, computational complexity, and algorithms. We will mention the basic techniques, and the relevant pointers to literature.

Mean-Payoff and Energy Games. The existence of memoryless winning strategies in mean-payoff games was established in [83], and the proof was based on induction on the number of edges and establishing the equivalence of the mean-payoff game played for finitely many steps and the mean-payoff game played forever. The algorithmic solution for mean-payoff games was given in [181], using a *value iteration* algorithm. Consider a sequence of *valuations* $(v_i)_{i\geq 0}$, where each valuation v_i is a function $v_i : S \to \mathbb{Z}$ defined as follows: (1) $v_0(s) = 0$ for all $s \in S$; and (2) for $i \geq 0$ we have

$$v_{i+1}(s) = \begin{cases} \max_{(s,t)\in E}\{w(s,t) + v_i(t)\} & \text{for } s \in S_1 \\ \min_{(s,t)\in E}\{w(s,t) + v_i(t)\} & \text{for } s \in S_2. \end{cases}$$

Observe that v_k can be computed in time $O(k \cdot m)$, where m is the number of edges. Let v_* be the optimal valuation of the mean-payoff game. The results of [181] show that

$$\frac{v_k}{k} - \frac{2 \cdot n \cdot W}{k} \leq v_* \leq \frac{v_k}{k} + \frac{2 \cdot n \cdot W}{k},$$

where W is the maximum absolute value of the weights. Furthermore, it was shown in [181] that by computing v_k for $k = 4 \cdot n^3 \cdot W$, the optimal value vector v_* can be computed. The result for energy games is similar: existence of memoryless winning strategies was established in [33], and a value iteration algorithm was also given. The running time of the value iteration algorithm is $O(n^3 \cdot m \cdot W)$. Recently, the value iteration algorithm has been improved by [28] to obtain an algorithm that runs in $O(n^2 \cdot m \cdot W)$ time. A strategy improvement algorithm for mean-payoff games is presented in [16]. We summarize the result in the following theorem (we present the result for mean-payoff objectives, but the result for energy objectives is similar).

Theorem 5 *Given a game graph G, with a weight function w,*

1. *For all $\theta \in \mathbb{N}$, we have $W_1(\mathsf{MeanPayoff}(\theta)) = S \setminus W_2(\mathsf{Plays}(G) \setminus \mathsf{MeanPayoff}(\theta))$ and memoryless winning strategies exist for both players.*
2. *Given a state s and $\theta \in \mathbb{N}$, whether $s \in W_1(\mathsf{MeanPayoff}(\theta))$ belongs to NP $\cap$ coNP.*
3. *For $\theta \in \mathbb{N}$, the winning set $W_1(\mathsf{MeanPayoff}(\theta))$ can be computed in $O(n^2 \cdot m \cdot W)$ time.*

Discounted Games. The existence of memoryless strategies in discounted games can be obtained as a special case of the result of Shapley [159]. The algorithm to solve discounted games is similar to the value iteration for mean-payoff games, and in discounted games the valuations need to be computed for $O(n^3 \cdot \frac{1}{1-\lambda})$ steps (see [92, 181] for details). The results for discounted games are as follows.

Theorem 6 *Given a game graph G, with a weight function w,*

1. *For all $\theta \in \mathbb{N}$ and rational $0 < \lambda < 1$, we have $W_1(\mathsf{Discounted}(\lambda, \theta)) = S \setminus W_2(\mathsf{Plays}(G) \setminus \mathsf{Discounted}(\lambda, \theta))$ and memoryless winning strategies exist for both players.*

2. *Given a state* s, *rational* $0 < \lambda < 1$, *and* $\theta \in \mathbb{N}$, *whether* $s \in W_1(\mathsf{Discounted}(\lambda, \theta))$ *can be decided in NP* $\cap$ *coNP.*
3. *For* $\theta \in \mathbb{N}$ *and rational* $0 < \lambda < 1$, *the winning set* $W_1(\mathsf{Discounted}(\lambda, \theta))$ *can be computed in* $O(n^3 \cdot m \cdot \frac{1}{1-\lambda})$ *time.*

27.2.6 Reducibility Between Graph Games

We now discuss the reducibility between various classes of games.

Parity to Mean-Payoff Games. A reduction of parity games to mean-payoff games was presented in [110]. The reduction is defined on the same game graph, and the reduction function is as follows: for a state with priority i, the reward is $(-1)^i \cdot n^i$, where n is the number of states. Then we have $W_1(\mathsf{Parity}(p)) = W_1(\mathsf{MeanPayoff}(0))$, i.e., the winning sets for parity and mean-payoff objectives coincide. The question of whether the decision problem for mean-payoff objectives can be reduced to parity objectives is open.

Mean-Payoff to Discounted Games. The reduction of mean-payoff games to discounted games was presented in [181]. The reduction was defined on the same game graph, with the same reward function, and the discount factor of λ defined as $1 - \frac{1}{4 \cdot n^3 \cdot W}$, where n is the number of states and W is the maximum absolute value of the weights. The question of whether the decision problem for discounted objectives can be reduced to mean-payoff objectives is open.

Energy Games and Mean-Payoff Games. The equivalence of the decision problem for finite initial credit for energy objectives and the mean-payoff objectives was established in [22]. The main argument is as follows: by the existence of memoryless strategies it follows that if the answer to the mean-payoff objectives with threshold $\theta = 0$ is true, then player 1 can fix a memoryless strategy such that in all cycles the sum of the rewards is non-negative, and this exactly coincides with the finite initial credit problem (where after a prefix, the sum of the rewards in cycles is non-negative). A similar argument holds for the reduction in the other direction.

27.2.7 Extensions

We briefly discuss several extensions of such games which have been studied in the literature, and give a few relevant references (there is no attempt to be exhaustive).

Stochastic and Concurrent Games. In this chapter we focused on games where the transitions are deterministic, and the games were turn-based (in each round one of the players makes a move). The class of *turn-based stochastic games* (games with

a probabilistic transition function) has been widely studied, for example in [34–36, 65, 77, 78]. The class of *concurrent* games where both players make their move simultaneously has also been studied in depth [5, 7, 37–39, 67, 91, 102, 127, 159]. For a survey of stochastic and concurrent games see [56].

Partial-Information Games. In *partial-information* games, the players choose their moves based on incomplete information about the state of the game. Such games are harder to solve than the corresponding perfect-information games. For example, turn-based deterministic (2-player) games with partial information and zero-sum reachability/safety objectives are EXPTIME-complete [149]. In the presence of more than two players, turn-based deterministic games with partial information and reachability objectives (for one of the players) are even undecidable [149]. A key technique to solve partial-information games (when possible) is reduction to perfect-information games, using a subset construction on the state space similar to the determinization of finite automata. The results in [54] present a close connection between a subclass of partial-information turn-based games and perfect-information concurrent games. The algorithmic analysis of partial-information stochastic games with ω-regular objectives has been studied in [44, 48, 49, 135]; the complexity of partial-information Markov decision processes has been studied in [40, 46, 136]. The more general class of partial-information stochastic games where both players have partial information has been studied in [14, 43]. Another interesting variety of partial-information games is the class of games where the starting state is unknown [103]. See [41, 47] for surveys related to partial-observation games.

Infinite-State Games. There are several extensions of games played on finite state spaces to games played on infinite state spaces. Notable examples are *pushdown* games and *timed* games. In the case of pushdown games, the state of a game encodes an unbounded amount of information in the form of the contents of a stack. Deterministic pushdown games are solved in [176] (see [178] for a survey); probabilistic pushdown games in [89, 90]; and pushdown games with quantitative objectives in [51, 68, 72]. In the case of timed games, the state of a game encodes an unbounded amount of information in the form of real-numbered values for finitely many clocks. Timed games are studied in [2, 123].

Quantitative and Qualitative Objectives. The problem of solving turn-based games with a conjunction of quantitative and qualitative objectives has been studied in [42, 60]; and multi-dimensional quantitative objectives in [27, 45, 71, 73, 174]. The problem of multi-dimensional objectives has also been widely studied for stochastic models [25, 26, 50, 66, 74, 87, 96].

Logical Framework for Games. Logical frameworks where properties for games can be described concisely with precise semantics for reasoning about games have also been studied in the literature. Some prominent examples of logical frameworks for reasoning about games are alternating-time temporal logic (ATL) and game logic [9]; strategy logic and various fragments [63, 130, 131]; and coordination logic [95].

27.3 Reactive Synthesis

27.3.1 Introduction

In this section, we summarize techniques to automatically construct reactive systems from specifications. A reactive system [124, 125] is a system that maintains an ongoing interaction with its environment. Examples of reactive systems are concurrent programs, air traffic control systems, controllers for mechanical devices, and digital hardware designs. We will use LTL as our representative specification language for concurrent systems (see Chap. 2 in this Handbook [139]). Many interesting properties such as mutual exclusion, deadlock freedom, fairness, and termination can be expressed in LTL.

We will limit ourselves to synchronous systems. This limitation implies that we consider games in which the players strictly alternate turns. The theory needed for asynchronous systems is somewhat different [115, 143, 145, 155, 170], as for such systems the interleaving of processes is not under the control of (or even known to) the system. Thus, it is not possible, for instance, to guarantee that the value of an output changes before a given input changes [1, 5]. (See also [79], where realizability is used to define the concept of "receptiveness" for asynchronous systems.)

The classical approach to synthesis (presented in Sect. 27.3.4) reduces the LTL synthesis problem to the problem of synthesizing a system that realizes a language defined by a Büchi automaton. Thus, this approach can be seen as a solution to the more general problem of deriving a system from a specification given as an ω-regular language.

Every synthesis problem consists of two inputs: (1) a specification that defines the desired behavior of the system and (2) a partition of variables used in the specification into input and output variables. Let us fix a set $\mathscr{I}$ of Boolean input signals and a set $\mathscr{O}$ of Boolean output signals. Thus, the input and output alphabet are $\Sigma_{\mathscr{I}} = 2^I$ and $\Sigma_{\mathscr{O}} = 2^O$, respectively, where a letter is a subset of $\Sigma_{\mathscr{I}} \cup \Sigma_{\mathscr{O}}$ that consists of those signals that are true (cf. Chap. 2 in this Handbook [139]). In LTL synthesis the specification is an LTL formula over a set of atomic propositions $\mathscr{I} \cup \mathscr{O}$.

The synthesis problem can be described as a turn-based game between two players: the environment and the system. In each round, the system picks an output from $\Sigma_{\mathscr{O}}$ and then the environment picks an input from $\Sigma_{\mathscr{I}}$, and the next round starts. (This order corresponds to a Moore machine, we will consider Mealy machines below.)

Example 2 Figure 2(a) gives a safety automaton for the specification $\square(r \rightarrow g \vee \bigcirc g)$—every request r must be followed by a grant g in the current or in the next step. States are represented by circles and transitions by arrows. Each transition is labeled with one or more conjuncts of atomic propositions or their negations (denoted by a bar), which indicate that a transition can only be taken with a letter that satisfies one of these conjuncts. A word is accepted by this safety automaton if its run stays within the accepting states (denoted by a double circle).

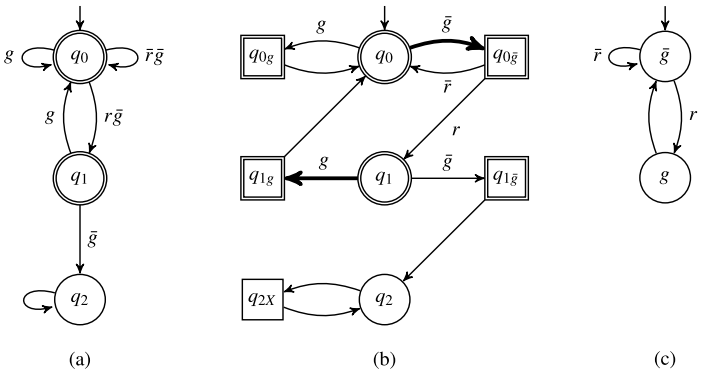


Fig. 2 A safety automaton, a labeled safety game, and a system that wins the game

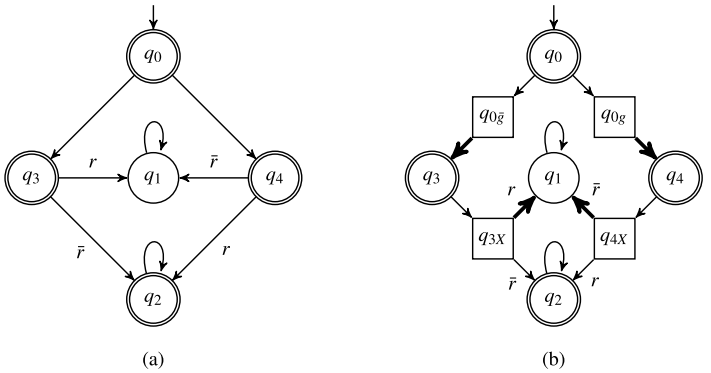


Fig. 3 A nondeterministic safety automaton, and a safety game that does NOT correspond to it

Figure 2(b) shows the safety game corresponding to this specification. The game is between the system (player 1, owner of the states depicted as circles), which controls output g, and the environment (player 2, owner of the boxes), which controls input r. The game is created by splitting every transition of the automaton into two parts: (i) one part controlled by the system and (ii) one part controlled by the environment. For instance, the self-loop on state q_0 with the label $\bar{r}\bar{g}$ in Fig. 2(a) is split into (i) the transition from state q_0 to $q_{0\bar{g}}$ in Fig. 2(b), which indicates that the system has chosen to set g to 0, and the transition from $q_{0\bar{g}}$ back to q_0, which indicates that the environment chooses to set r to 0. The winning condition for player 1 mirrors the acceptance condition of the automaton: stay within the accepting states.

Any player-1 strategy that follows the specification is winning for this game. The bold arrows in Fig. 2(b) indicate one such winning strategy. Since strategies on safety games are memoryless (see Theorem 1), a correct system can be implemented by keeping track of the state of the game and always playing the proper response, as shown in Fig. 2(c).

For the acceptance conditions that we consider, this simple transformation works as long as the specification is given by a *deterministic* automaton. For nondeterministic automata it does not work: the nondeterministic automaton in Fig. 3 accepts any word, but the game in the same figure on the right is lost for player 1. (The bold arrays indicate a winning strategy for player 2, which shows that player 1 cannot win this game from the initial state.) The need for determinization has a significant impact on complexity, as we will see later.

In order to define the LTL synthesis problem formally, we first give a formal definition of transducers, which describe the desired systems. We refer the reader to Chap. 2 in this Handbook [139] for a detailed description of LTL.

27.3.2 Games, Transducers, Trees, and Automata

In the following, we define labeled games as deterministic tree automata. The two formalisms are equivalent. We will need universal tree automata as an intermediate step between a logical specification and the resulting transducer. The relation between (nondeterministic) tree automata and games without labels is formalized in [101].

Note that the complete behavior of a transducer is a tree, where nodes are labeled with outputs and edges with inputs: the output of the transducer after input word w is the label of the node at the end of the path labeled w. Thus, a tree automaton defines a set of transducers. In the following, we will formalize this notion.

Definition 1 (Tree, Labeled Tree) Given a finite alphabet D of directions, a *D-tree* $T \subseteq D^*$ is a prefix-closed set of words over D. The nodes $v \cdot d$ for $d \in D$ are the *children* of v; v is their *parent*. The empty word ε is called the *root* of T.

A *path* ρ of T is a prefix-closed subset of T such that (1) the root is in ρ (i.e., $\varepsilon \in \rho$), and (2) every node has at most one child (i.e., $\forall v \in T\, \forall d_1, d_2 \in D$, if $d_1 \neq d_2$ and $v \cdot d_1 \in \rho$, then $v \cdot d_2 \notin \rho$). A tree T is *complete* if $T = D^*$.

A *Σ-labeled D-tree* is a pair (T, τ), where T is a tree and $\tau : T \to \Sigma$ is a *labeling function* mapping every node in T to a letter from a finite *alphabet* Σ.

Definition 2 (Transducer) A *(finite-state Moore) transducer* is a tuple $\mathcal{M} = \langle \mathcal{I}, \mathcal{O}, M, m_0, \alpha, \lambda \rangle$, where M is a (finite) set of *states*, $m_0 \in M$ is the *initial state*, $\alpha : M \times \Sigma_{\mathcal{I}} \to M$ is a *transition function* mapping a state and an input to a successor state, and $\lambda : M \to \Sigma_{\mathcal{O}}$ is a *labeling function* that maps every state to an output.

We extend α from input letters to input words in the usual way, i.e., $\alpha(m, \varepsilon) = m$ and $\alpha(m, v_0 \dots v_n) = \alpha(\alpha(m, v_0 \dots v_{n-1}), v_n)$. An *execution* of $\mathcal{M}$ on input $x_0, x_1, \dots \in \Sigma_{\mathcal{I}}$ is a word $m_0, m_1, \dots$, where $m_i = \alpha(m_{i-1}, x_{i-1})$ for all $i > 0$. The associated *I/O word* is $\lambda(m_0) \cup x_0, \lambda(m_1) \cup x_1, \dots$, and $\mathcal{L}(\mathcal{M})$ is the set of all I/O words of $\mathcal{M}$.

Every transducer $\mathscr{M} = \langle \mathscr{I}, \mathscr{O}, M, m_0, \alpha, \lambda \rangle$ generates a complete $\Sigma_{\mathscr{O}}$-labeled $\Sigma_{\mathscr{I}}$-tree (T, τ) with $\tau(\varepsilon) = \lambda(m_0)$ and $\tau(v_0 \dots v_n) = \lambda(\alpha(m_0, v_0 \dots v_{n-1}))$. We denote by $\mathscr{L}_T(\mathscr{M})$ the singleton set that contains this tree. A complete labeled tree (T, τ) is called *regular* if there exists a finite-state transducer that generates (T, τ).

Definition 3 (Deterministic Tree Automaton) A *deterministic tree automaton (on infinite labeled trees)* is a tuple $\mathscr{A} = \langle D, \Sigma, Q, q_0, \delta, \phi \rangle$, where D and Σ are a finite set of *directions* and *letters*, respectively, Q is a finite set of *states*, $q_0 \in Q$ is the *initial state*, $\delta : Q \times \Sigma \to D \to Q$ is the *transition function*, and ϕ is an *acceptance condition* that specifies a subset of Q^ω.

Given a $\Sigma_{\mathscr{O}}$-labeled $\Sigma_{\mathscr{I}}$-tree (T, τ), a *run* of a deterministic tree automaton $\mathscr{A}$ on (T, τ) is an isomorphic Q-labeled tree (T, τ_r) in which (1) $\tau_r(\varepsilon) = q_0$ and (2) if $\tau_r(v) = q$ and $\tau(v) = \sigma$, then $\tau_r(v \cdot d) = \delta(q, \sigma, d)$. (The acceptance conditions correspond to the winning objectives defined in Sect. 27.2.2. We will discuss them more below.)

Example 3 The labeled game in Fig. 2(b) is formally defined by a deterministic tree automaton with $D = \{\emptyset, \{r\}\}$, $\Sigma = \{\emptyset, \{g\}\}$, $Q = \{q_0, q_1, q_2\}$, $\delta(q_0, \emptyset) = \{(\emptyset, q_0), (\{r\}, q_1)\}$,[2] $\delta(q_0, \{g\}) = \{(\emptyset, q_0), (\{r\}, q_0)\}$, $\delta(q_1, \emptyset) = \cdots$, and $\phi = \{q_0, q_1\}^\omega$. The player-1 states (circles) correspond to states of the automaton; the player-2 states (boxes) can be seen as the different transitions of the automaton, i.e., pairs of states and letters. Note that the automaton is deterministic, because in every state for every pair of letters and directions, there exists exactly one successor state.

As an intermediate step in some synthesis procedures, we will use universal tree automata. They differ from deterministic tree automata by being able to send multiple copies of the automaton, in different states, to a child of a tree node.

Definition 4 (Universal Tree Automaton) A *universal tree automaton* is a tuple $\mathscr{A} = \langle D, \Sigma, Q, q_0, \delta, \phi \rangle$, where $\delta : Q \times \Sigma \to 2^{D \times Q}$ is the *transition function*, and everything else is defined as for deterministic tree automata.

Deterministic tree automata are a special case of universal tree automata. Runs of universal automata, however, are more complicated: they are not isomorphic to the input tree. Thus, we label each node of the run tree with the node of the input tree to which it pertains.

Note that the relation between universal and deterministic automata is more complicated for infinitary acceptance conditions than in the finitary case, even for word automata. By symmetry, the same holds for the relation between nondeterministic and deterministic automata. For instance, not every Nondeterministic Büchi Word

[2] We use a set of tuples $D \times Q$ to represent a function $D \to Q$.

automaton has an equivalent Deterministic Büchi Word automaton [167] (and symmetrically, a Universal co-Büchi automaton cannot always be translated to Deterministic co-Büchi Word automaton). In other cases, the construction may be possible but very complicated, as with the determinization of parity automata [137, 151], or it may be close to the well-known subset construction (as for the translation of Universal Büchi automata to Deterministic Büchi automata) [129]. As we will see later, the complexity of the determinization procedure for parity automata is an important reason that the standard approach to synthesis is quite expensive.

Given a $\Sigma_{\mathscr{O}}$-labeled $\Sigma_{\mathscr{I}}$-tree (T, τ), a run of a universal tree automaton $\mathscr{A}$ on (T, τ) is a $T \times Q$-labeled tree (T_r, τ_r) in which (1) $\tau_r(\varepsilon) = (\varepsilon, q_0)$ and (2) for any $v \in T_r$, if $\tau_r(v) = (n, q)$, $\tau(n) = \sigma$, and $\delta(q, \sigma) = \{(d_1, q_1), \ldots, (d_n, q_n)\}$, then v has children $v_1, \ldots, v_n$ labeled $(n \cdot d_1, q_1), \ldots, (n \cdot d_n, q_n)$. Note that branches of the run tree can be finite if $\delta(q, \sigma) = \emptyset$.

Acceptance Condition. A run (T_r, τ_r) is *accepting* if all its *infinite* paths ρ satisfy the acceptance condition. The acceptance condition ϕ on paths is defined in the same way as winning objectives on plays (see Sect. 27.2.2). For example, a *Büchi condition* is given by a set $B \subseteq Q$ of target states and we define ϕ to be all the paths on which we see infinitely often a state from B, i.e., $\phi = \{q_0 q_1 \cdots \in Q^\omega \mid \forall i \geq 0 \exists j > i, q_j \in B\}$. The language $\mathscr{L}_T(\mathscr{A})$ of $\mathscr{A}$ is the set of all trees t such that the run of $\mathscr{A}$ on t is accepting. Note that finite paths can never be a reason for rejection.

If $|D| = 1$, then $\mathscr{A}$ is called a *word automaton*. In this chapter, we are not interested in nondeterministic and alternating tree automata [86, 134]. Automata types are typically denoted by three letter acronyms, where the first letter denotes the branching (A for alternating, U for universal, N for nondeterministic, or D for deterministic), the second letter describes the acceptance condition (B for Büchi, C for co-Büchi, or P for parity), and the third letter is T for tree automata or W for word automata.

Example 4 We will ignore Fig. 4 for now. Figure 5 shows a universal co-Büchi tree automaton (UCT) with letters $\Sigma = 2^{\{g_1, g_2\}}$ (two grant signals) and directions $D = 2^{\{r_1, r_2\}}$ (two request signals). Recall that a UCT accepts a tree if none of its paths visits a co-Büchi state infinitely often (cf. Sect. 27.2.2). We show part of an input tree and the corresponding run in Fig. 6. Consider the infinite path indicated with dashed bold lines Fig. 6. The sequence of directions along this path is $\{r_1 r_2\}, \emptyset, \emptyset, \{r_1 r_2\}, \emptyset, \emptyset, \ldots$, which captures the behavior in which the environment sends two requests in every third step.

This path of the input tree is labeled with the following output letter sequence $\{g_1\}, \emptyset, \{g_2\}, \{g_1\}, \emptyset, \{g_2\}$; the sequence states that the system responds to this input behavior by the following three-step pattern: first it sets g_1 to high, then it lowers both grants, and finally it sets g_2 to high.

On the right of Fig. 6 we depict the corresponding part of the run of the UCT on the input tree. Initially, the automaton is in state q_0 and it reads the label of the root of the tree (i.e., $\{g_1\}$), which enables transition t_1. From t_1 we have to move according to the direction that we consider. In our example this direction is $\{r_1 r_2\}$,

Fig. 4 NBW for $\neg\varphi$

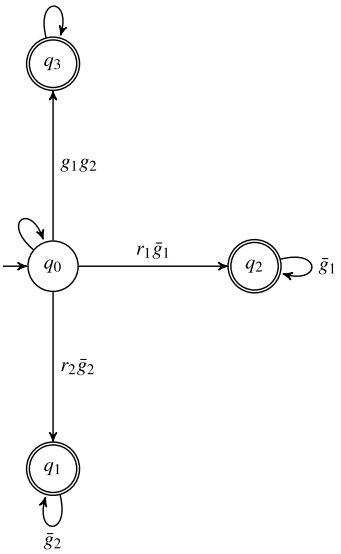
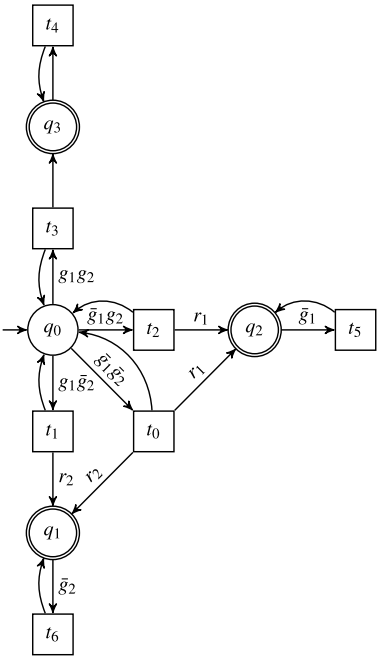


Fig. 5 UCT for φ



which enables the edge from t_1 to q_0 *and* the edge from t_1 to q_1. Since the automaton has universal branching mode, we have to follow both edges and the run continues in both states. From state q_0 with label $\emptyset$ (input tree node: 3), the automaton has to select transition t_0 and the direction $\emptyset$ leads again to q_0. From state q_1 the label $\emptyset$

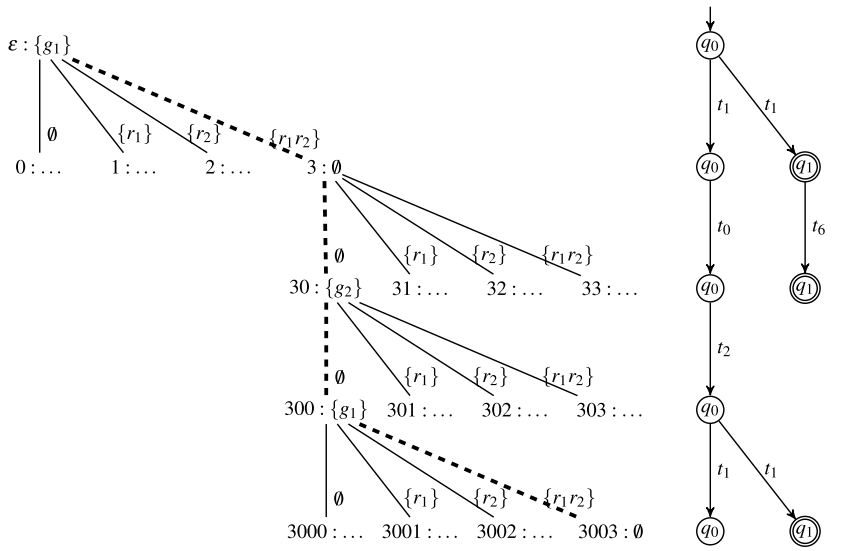


Fig. 6 (*Left*) Part of an input tree. (*Right*) UCT-run on the dashed path of the input tree

enables transition t_6, which brings us back to q_1. When the automaton is in state q_1 and it reaches node 30 labeled $\{g_2\}$ no transition is enabled, therefore this path of the run ends here. Recall that a run is accepting if none of the paths visits a co-Büchi state (indicated by a double circle) infinitely often, and a finite path is always accepting. In Fig. 6, we show only the first part of the input tree and part of the corresponding run. The depicted part of the run will repeat as the input repeats.

27.3.3 Realizability and Synthesis Problem

Given an LTL formula φ over $\mathscr{I} \cup \mathscr{O}$, and a transducer $\mathscr{M} = \langle \mathscr{I}, \mathscr{O}, M, m_0, \alpha, \lambda \rangle$, we say that $\mathscr{M}$ *realizes (or implements)* φ if $\mathscr{L}(\mathscr{M}) \subseteq \mathscr{L}(\varphi)$.

Definition 5 (LTL Realizability and Synthesis Problem) Given an LTL formula φ over the atomic propositions $\mathscr{I} \cup \mathscr{O}$, the *realizability problem* asks whether there exists a transducer $\mathscr{M}$ that realizes φ. If the answer to the realizability problem is yes, then we call the specification φ *realizable*. The *synthesis problem* is to construct $\mathscr{M}$.

Let us make two simple observations: First, constructing a Mealy machine is equally easy (or hard) as constructing a Moore machine. It can be achieved by shifting inputs by one time step. Taking LTL as an example, ϕ is Mealy-realizable iff ϕ' is Moore-realizable, where ϕ' is obtained from ϕ by replacing every occurrence of an output signal y by $\bigcirc y$. Equivalently, one can switch the order of the players in the corresponding game. Second, when negation is possible and the game is

determined (as with LTL), then ϕ is Mealy-realizable with inputs $\mathscr{I}$ and outputs $\mathscr{O}$ iff $\neg\phi$ is Moore-realizable with inputs $\mathscr{O}$ and outputs $\mathscr{I}$. In other words, there is a system that fulfills the specification iff there is no environment that guarantees violation (and vice versa).

27.3.4 Classical Approach to LTL Synthesis

In this section we summarize the classical approach to LTL synthesis [30, 142, 146]. The approach consists of the following steps:

1. Translate the LTL formula φ into a nondeterministic Büchi word automaton A.
2. Translate A into a deterministic parity word automaton (DPW) $\mathscr{B}$.
3. Construct a deterministic parity tree automaton (DPT) $\mathscr{A}_T$ from $\mathscr{B}$.
4. Check language emptiness of $\mathscr{A}_T$ (i.e., solve the parity game).
5. If $\mathscr{A}_T$ is non-empty, construct a finite-state transducer $\mathscr{M}$, otherwise report that φ is not realizable.

Example 5 (Arbiter Example) We use a specification for a simple arbiter to show the approach. The arbiter controls the access of two clients, C_1 and C_2, to a shared resource. It has two input variables r_1 and r_2 and two output variables g_1 and g_2. Client i can request the resource by setting the input variable r_i to true. The arbiter grants the resource to Client i by setting the corresponding output variable g_i to true.

We require the arbiter to ensure (i) *mutually exclusive* and (ii) *fair* access to the resource. Formally, the specification $\varphi_A = \psi \wedge \varphi_1 \wedge \varphi_2$ is the conjunction of the following three properties:

$$\begin{aligned}
\psi &= \square(\neg g_1 \vee \neg g_2) && \text{mutually exclusive access of the clients,}\\
\varphi_1 &= \square(r_1 \rightarrow \Diamond g_1) && \text{fair access for Client 1, and}\\
\varphi_2 &= \square(r_2 \rightarrow \Diamond g_2) && \text{fair access for Client 2.}
\end{aligned}$$

Step 1: LTL to NBW

Given an LTL formula φ, we first construct a nondeterministic Büchi word automaton $\mathscr{A}_\varphi$ such that $\mathscr{A}_\varphi$ accepts all the words that satisfy φ, i.e., $\mathscr{L}(\varphi) = \mathscr{L}(\mathscr{A}_\varphi)$ with $|A_\varphi| = 2^{O(|\varphi|)}$ [171]. (Several people have worked on improving this translation, e.g., [88, 99, 157, 158, 165].)

Step 2: NBW to DPW

Using Piterman's determinization construction [137] (an improved version of Safra's construction [151]), we translate $\mathscr{A}$ into a deterministic Parity word automaton $\mathscr{B}$ such that $\mathscr{L}(\mathscr{A}) = \mathscr{L}(\mathscr{B})$. This automaton has $2^{2^{O(|\varphi|)}}$ states and $2^{O(|\varphi|)}$ priorities.

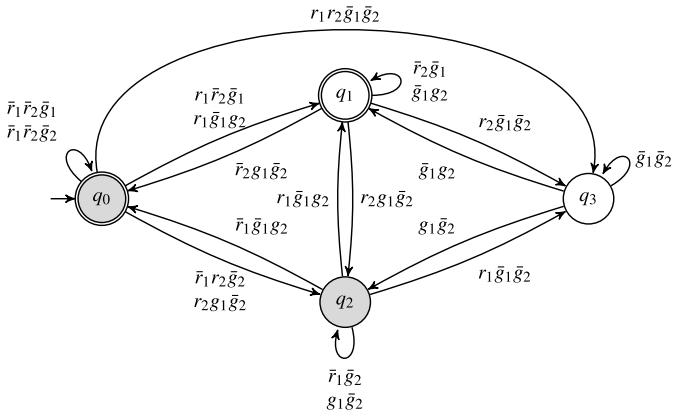


Fig. 7 Deterministic Streett word automaton for φ_A with the Streett pairs $R = \{(Q, \{q_0, q_1\}), (Q, \{q_0, q_2\})\}$

Example 6 (Arbiter Example (Cont.)) For simplicity, we show in Fig. 7 a deterministic Streett (instead of parity) word automaton for the specification φ_A. The winning condition is that sets $\{q_0, q_1\}$ and $\{q_0, q_2\}$ (marked with double circles and filled circles, respectively) must be visited infinitely often (a *generalized Büchi condition*). Formally, the automaton $\mathscr{A}_S$ has two Streett pairs $(Q, \{q_0, q_1\})$ and $(Q, \{q_0, q_2\})$. The intuitive meaning of the four states q_0, q_1, q_2 and q_3 is as follows: there are no outstanding requests in state q_0. There is an outstanding request from Client 1 (or Client 2) in state q_1 (or q_2, respectively). In state q_3 both requests are outstanding.

Step 3: DPW to DPT

It is easy to convert the DPW $\mathscr{B}$ obtained in Step 2 to a DPT $\mathscr{A}_\varphi^T$ such that $\mathscr{A}_\varphi^T$ accepts a tree iff all its paths satisfy φ. Intuitively, we split the transitions into two parts: the first part refers to the output variables (the alphabet of the tree automaton), the second part to the input variables (the directions of the tree automaton).

Example 7 (Arbiter Example (Cont.)) Figure 8 shows the tree automaton generated from the automaton in Fig. 7. The construction was described in Example 2, and will not be formalized.

Step 4: DPT Emptiness Check

The language of $\mathscr{A}_\varphi^T$ is non-empty iff it contains a $\Sigma_{\mathscr{O}}$-labeled $\Sigma_{\mathscr{I}}$-tree, i.e., iff player 1 (the system) has a winning strategy in the corresponding game. (See Example 3 for the correspondence between DPTs and games.) We can use the techniques describes in Sect. 27.2 to check whether $\mathscr{A}_\varphi^T$ (a parity game) is empty.

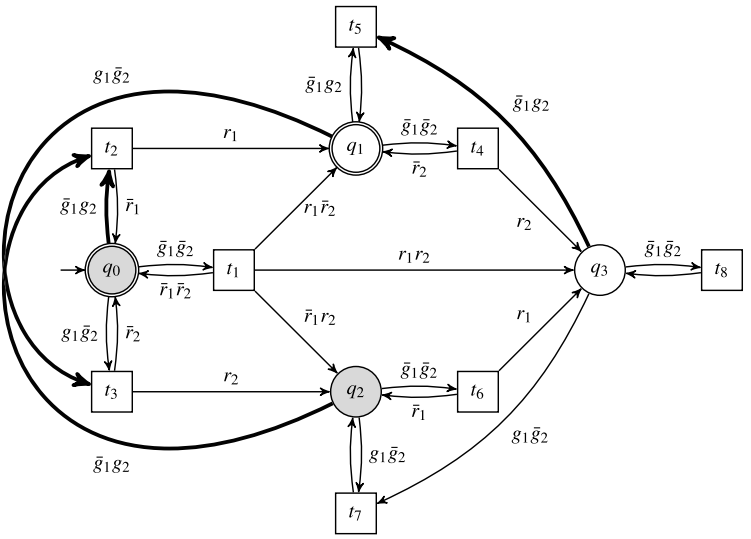


Fig. 8 Streett tree automaton for φ_A (generated from the automaton in Fig. 7). The *bold arrows* denote a winning (memoryless) strategy for player 1 in the corresponding Streett game

In general, the LTL formula is translated into a parity game with $2^{2^{O(|\varphi|)}}$ states and $2^{O(|\varphi|)}$ priorities, which can be solved in polynomial time in the number of states and exponential time in the number of priorities (see Theorem 3). A corresponding doubly exponential lower bound for LTL synthesis was shown by Rosner [150].

Step 5: Construction of Finite-State Transducer

If the language of the deterministic parity tree automaton $\mathcal{A}_\varphi^T$ is non-empty, then there exists a winning memoryless strategy for player 1 in the corresponding parity game. The strategy corresponds to a regular tree, and regular trees coincide with finite-state transducers.

Example 8 (Arbiter Example (Cont.)) The winning objective of player 1 is to visit the set $\{q_0, q_2\}$ and $\{q_0, q_1\}$ infinitely often. The bold arrows in Fig. 8 denote a memoryless winning strategy for player 1 in the corresponding game, where states $q_0, \ldots, q_3$ correspond to player-1 states of the game and the transitions $t_1, \ldots, t_8$ are the player-2 states. Note that memoryless strategies suffice for parity games. (For Streett games, which we use in the example, memoryless strategies are not sufficient, see Sect. 27.2.2.)

The strategy corresponds to the finite-state transducer shown in Fig. 9 with the three states q_0, q_1 and q_2 labeled with $\{g_2\}$, $\{g_1\bar{g}_2\}$, and $\{g_2\}$, respectively. Following the strategy, the transducer initially outputs g_2, and then moves to q_0 or q_1 depending on the input. If Client 1 sends a request (i.e., r_1 is high), then the transducer

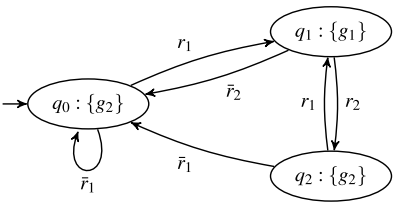


Fig. 9 Finite-state transducer implementing the specification φ_A

moves to state q_1 and sends a grant to Client 1 (i.e., it outputs g_1). Otherwise, the arbiter grants the shared resource to Client 2 by raising g_2. State q_3 is not reachable with the given strategy.

Theorem 7 ([30, 142, 146]) *Given an LTL formula φ, we can decide in $2^{2^{O(|\varphi|)}}$ time whether φ is realizable. If an LTL formula φ is realizable, then there exists a finite-state transducer with at most $2^{2^{O(|\varphi|)}}$ states that satisfies it. Both these bounds are tight.*

27.3.5 Recent Approaches to LTL Synthesis

We describe two main ideas to cope with the complexity of the LTL synthesis problem: (i) **bounding** the size of the **generated systems** and (ii) specialized procedures for **specification with restricted expressiveness**, and combinations thereof. Recent approaches are based on one or both ideas.

27.3.5.1 Bounded (or Safraless) Approaches

The idea of bounded synthesis approaches [93, 120, 156] is to restrict the size of the generated system. These algorithms are also called *Safraless*, because they avoid Safra's determinization construction. Bounded synthesis algorithms are based on the following two key insights, which were first presented by Kupferman and Vardi [120].

1. The LTL synthesis problem can be reduced to the language emptiness check of a universal co-Büchi tree automaton (UCT).
2. The language emptiness problem of UCTs can be reduced to a parametric emptiness check, where the parameter restricts the size of the trees of interest (and hence the size of the generated system).

In [120], the emptiness problem of a universal co-Büchi tree automaton with parameter k (k-UCT) is reduced to the emptiness problem of a nondeterministic Büchi tree automaton. Checking emptiness of a nondeterministic Büchi tree automaton in turn corresponds to solving a Büchi game [101]. For the purpose of this paper, we can assume that k limits the number of times that the automaton can visit a co-Büchi

state [156]. In [156], Schewe and Finkbeiner present a reduction of the k-UCT emptiness problem to emptiness of deterministic safety tree automata. The exact meaning of k is different in [120], but the general idea is the same. These bounded approaches are well suited for symbolic implementations. Schewe and Finkbeiner propose an encoding as an SMT formula, while Filiot, Jin, and Raskin [93] provide a symbolic algorithm for checking k-UCT emptiness using anti-chains. (see also [82]). Bounded approaches can also be used as a semi-decision procedure for distributed synthesis [156].

A bounded synthesis algorithm consists of the following steps:

1. Translate the LTL formula φ into a UCT $\mathscr{A}$.
2. Transform $\mathscr{A}$ into a k-UCT $\mathscr{A}_k$ for a given parameter k.
3. Check emptiness of $\mathscr{A}_k$ (solving a Büchi or Safety game).
4. If $\mathscr{A}_k$ is non-empty, construct an FSM $\mathscr{M}$, otherwise increase k and go to Step 2 or abort.

For realizable specifications the parameter k turns out to be small in practice [93, 108], leading to an efficient LTL synthesis procedure if the specification is realizable. In order to conclude that a specification is unrealizable, we must show that $\mathscr{A}_k$ is empty for a very large k (doubly exponential in the size of the formula φ, see Theorems 8 and 9). However, it follows from the remarks in Sect. 27.3.3 that in case of unrealizability there is a Mealy machine for the environment that realizes $\neg\phi$ which can be found in much the same manner as the Moore machine for the system. Again, in practice this machine can be quite small and is then found quickly.

Step 1: LTL to UCT

Given an LTL formula and a partitioning of the atomic propositions, we can construct a universal co-Büchi tree automaton that accepts all state machines that realize the formula. Intuitively, we construct an NBW for the negated formula (see Step 1 in Sect. 27.3.4). We then dualize the acceptance condition and the branching condition and transform the automaton into a tree automaton as described above.

Theorem 8 ([120], Theorem 5.1) *The realizability problem for an LTL formula can be reduced to the non-emptiness problem for a UCT with exponentially many states.*

Example 9 (Arbiter Example (Cont.)) Recall the specification of the arbiter from Example 5: $\varphi = \Box(\neg g_1 \vee \neg g_2) \wedge \Box(r_1 \rightarrow \Diamond g_1) \wedge \Box(r_2 \rightarrow \Diamond g_2)$. We can construct a UCT for φ by first constructing an NBW for the negation of the specification, i.e., $\neg\varphi = \Diamond(g_1 \wedge g_2) \vee \Diamond(r_1 \wedge \Box\neg g_1) \vee \Diamond(r_2 \wedge \Box\neg g_2)$. Note that $\neg\varphi$ simplifies to $\Diamond((g_1 \wedge g_2) \vee (r_1 \wedge \Box\neg g_1) \vee (r_2 \wedge \Box\neg g_2))$ for which we show an NBW in Fig. 4. The automaton accepts a word in one of the following cases. Each case corresponds to violating the specification in a particular way: (i) Any word that includes simultaneous grants at some point will be accepted. In this case the automaton stays in state

q_0 until $g_1 g_2$ occurs and then moves to state q_3, which is accepting and can be revisited independently of the values of the grant and request signals. (ii) A word that includes a request 1 but no subsequent grant 1 is accepting. In order to see this, note that the automaton again can stay in state q_0 until the unanswered request 1 arrives, then it moves to state q_2, where it can stay as long as it does not observe a grant 1 (i.e., as long as the request is unanswered). (iii) The situation for an outstanding request 2 is analogous. In this case the automaton will move to state q_1.

Given this NBW and a splitting of the atomic propositions into inputs r_1, r_2 and output propositions g_1, g_2, we construct the UCT shown in Fig. 5. The alphabet of the tree automaton is $2^{\{g_1,g_2\}}$; each tree has four directions corresponding to the letters in $2^{\{r_1,r_2\}}$. The dualization of the branching mode means that nondeterministic edges are now viewed as universal edges. Dualizing the Büchi condition results in a co-Büchi acceptance condition.

Step 2: UCT to k-UCT

We will present the reduction from UCT emptiness to emptiness of deterministic safety tree automata, which corresponds to finding the winning player in games with safety objectives. The reduction to Büchi games can be found in [120], Theorem 3.3. Note that in the reduction to Büchi games the parameter k is used in a different way than in the reduction presented here.

The reduction from UCT emptiness to safety games is best explained in two steps. In the first step, we reduce the UCT emptiness problem to the emptiness problem of a tree automaton with a simpler *universal k-co-Büchi* acceptance condition, which asks that every path of an accepting run visits a co-Büchi state at most k times [156]. In the second step, we show how to check whether the language of a universal k-co-Büchi tree automaton is empty by constructing an equivalent safety game and solving it.

Given a UCT $\mathscr{A}$, we write $\mathscr{A}_k$ to denote the tree automaton with the same structure as $\mathscr{A}$ and the universal k-co-Büchi acceptance condition. For any UCT $\mathscr{A}$ and any parameter k, $\mathscr{L}_T(\mathscr{A}_k) \subseteq \mathscr{L}_T(\mathscr{A})$ holds, so if the language of $\mathscr{A}_k$ is non-empty, then so is the language of $\mathscr{A}$.

For the other direction, we will use the following two lemmas. The first one shows that if the language of the UCT $\mathscr{A}$ is non-empty, then there exists a finite-state machine of bounded size that is accepted by $\mathscr{A}$. The second states that a given finite-state machine is accepted by automaton $\mathscr{A}$ if and only if it is accepted by the automaton $\mathscr{A}_k$, where $k = |\mathscr{M}| \cdot |\mathscr{A}|$.

Lemma 1 ([120], Theorem 4.3) *Given a UCT $\mathscr{A}$ with n states, if the language of $\mathscr{A}$ is not empty, i.e., $\mathscr{L}_T(\mathscr{A}) \neq \emptyset$, there exists a (non-empty) finite-state machine $\mathscr{M}$ with at most $n^{n+1} + 1$ states such that $\mathscr{L}_T(\mathscr{M}) \subseteq \mathscr{L}_T(\mathscr{A})$ (meaning that the tree generated by $\mathscr{M}$ is accepted by $\mathscr{A}$).*

Lemma 2 ([156]) *Given a finite-state machine $\mathscr{M}$ and a UCT $\mathscr{A}$, then $\mathscr{A}$ accepts $\mathscr{M}$ if and only if the k-UCT $\mathscr{A}_k$ with $k = |\mathscr{M}| \cdot |\mathscr{A}|$ accepts $\mathscr{M}$, i.e., $\mathscr{L}_T(\mathscr{M}) \subseteq \mathscr{L}_T(\mathscr{A})$ iff $\mathscr{L}_T(\mathscr{M}) \subseteq \mathscr{L}_T(\mathscr{A}_{|\mathscr{M}|\cdot|\mathscr{A}|})$ holds.*

From Lemmas 1 and 2, it follows that if the language of a UCT $\mathscr{A}$ of size n is non-empty, then so is the language of the k-UCT $\mathscr{A}_k$ with $k = (n^{n+1}+1) \cdot n$ and we can obtain the following theorem.

Theorem 9 ([156]) *For any UCT $\mathscr{A}$ of size n, there exists a k-UCT $\mathscr{A}_k$ with $k = (n^{n+1}+1) \cdot n$ such that $\mathscr{L}_T(\mathscr{A}) = \emptyset$ iff $\mathscr{L}_T(\mathscr{A}_k) = \emptyset$.*

Note that a UCT and a k-UCT differ only in the interpretation of the acceptance condition. A UCT accepts all trees that allow a run on which none of the paths visits rejecting states infinitely often. A k-UCT is more restricted; it allows at most k visits to the rejecting states. So, Fig. 5 can be seen as a UCT or as a k-UCT.

Step 3: k-UCT Emptiness Check

Given a k-UCT, we can construct a safety game with labels (i.e., a deterministic safety tree automaton) such that the safety game is winning if and only if the language of the k-UCT is not empty.

Lemma 3 ([156]) *Given a k-UCT $\mathscr{A}$ with n states, there exists a deterministic safety tree automaton $\mathscr{B}$ with $(k+2)^n$ states such that $\mathscr{L}_T(\mathscr{A}) = \mathscr{L}_T(B)$.*

This lemma can be proven as follows. Intuitively, $\mathscr{B}$ is constructed from $\mathscr{A}$ by applying an extended subset construction that keeps track of how many rejecting states (i.e., states in F) have been visited so far along all the paths ending in a state of $\mathscr{A}$. To do that, $\mathscr{B}$ has a counter for every state in $\mathscr{A}$ that counts from -1 up to $k+1$. A state of $\mathscr{B}$ is an evaluation of all the counters. Counter value -1 of a state q means that state q is not reached in the current step of the subset construction. Formally, let $\mathscr{A} = \langle 2^{\mathscr{I}}, 2^{\mathscr{O}}, Q, q_0, \delta, F\rangle$, then $\mathscr{B} = \langle 2^{\mathscr{I}}, 2^{\mathscr{O}}, S, s_0, \delta_{\mathscr{B}}, F_{\mathscr{B}}\rangle$ is given by

$$S = Q \to \{-1, 0, \ldots, k+1\}$$

$$s_0(q) = \begin{cases} 0 & \text{if } q = q_0 \\ -1 & \text{otherwise.} \end{cases}$$

$$\delta_{\mathscr{B}}(s, o) = \bigwedge_{i \in 2^{\mathscr{I}}} (i, s'), \quad \text{where}$$

$$s'(q) = \begin{cases} k+1 & \text{if } \exists p : (i,q) \in \delta(p,o) \wedge s(p) = k+1, \\ \max_{p \in Q:(i,q)\in\delta(p,o)}\{s(p)+1\} & \text{if } \exists p : (i,q) \in \delta(p,o) \wedge -1 < s(p) < k+1 \wedge \\ & \quad q \in F, \\ \max_{p \in Q:(i,q)\in\delta(p,o)}\{s(p)\} & \text{if } \exists p : (i,q) \in \delta(p,o) \wedge -1 < s(p) < k+1 \wedge \\ & \quad q \notin F, \\ -1 & \text{otherwise } (\forall p : (i,q) \notin \delta(p,o) \vee s(p) = -1). \end{cases}$$

$$F_{\mathscr{B}} = \{s \in S \mid \forall q \in Q : s(q) < k+1\}$$

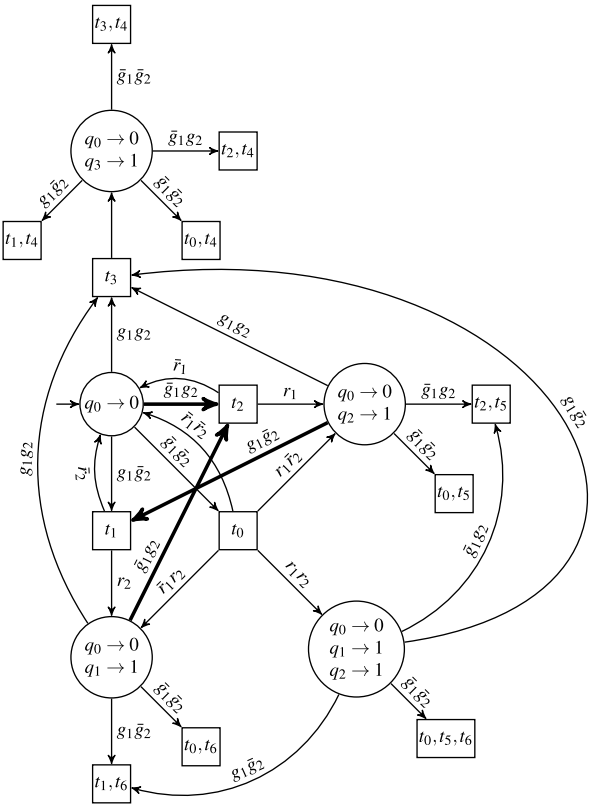


Fig. 10 Safety game (with labels) for the UCT shown in Fig. 5, with $k = 1$. The *bold arrows* represent a winning strategy from all winning states for player circle

Note that for each label $o \in 2^{\mathscr{O}}$, the automaton $\mathscr{B}$ has exactly one successor for each direction $i \in 2^{\mathscr{I}}$.

Example 10 (Arbiter Example (Cont.)) Recall the UCT from Fig. 5. Assume we fix $k = 1$, i.e., for every path we allow at most one visit to a rejecting state. In each player-1 state of the safety game we store two pieces of information: (i) the set of active states (of the UCT) in the current run and (ii) the number of rejecting states we have seen along the way. As shown after Lemma 3 we can represent this information by a function mapping from the UCT states to a number between -1 and k, i.e., the state $(q_0 \to 0, q_1 \to 2, q_2 \to -1, q_3 \to -1)$ indicates that the run is currently in state q_0 and q_1 and that on the path to q_1 we have seen two rejecting states (see the third level of the run on the right of Fig. 6). Figure 10 shows the safety game for the UCT shown in Fig. 5 with $k = 1$. In order to save space we omit UCT-states that are assigned to -1 in the description of the player-1 states. For simplicity, we label player-2 states with transitions of the UCT. In order to keep the game graph small, we also omit "unsafe" states, which are states in which a UCT-state is mapped to a number higher than k. This leads to several player-2 states without outgoing edges, called dead-end states. In all these states, player 2 wins because he could move to an

unsafe state. So, the (safety) winning condition for player 1 in this game is to avoid these dead-end states.

The game starts in state $(q_0 \to 0)$. The system (player 1) has four options corresponding to the four output letters. If she chooses the letter $\bar{g}_1 g_2$ the play continues in state t_2, from which player 2 can choose one of the four directions. If he chooses a direction that includes a request to Client 1 (r_1), then the play moves to state $(q_0 \to 0, q_2 \to 1)$. Note that on the path to state q_2, we have seen one rejecting state (namely q_2 itself), therefore q_2 is mapped to 1. Intuitively, q_2 indicates that there is an outstanding request from Client 1, so staying in q_2 forever violates the specification. Since we have chosen $k = 1$, we are only allowed to visit q_2 for one step (which corresponds to delaying the grant by one step). So, from state $(q_0 \to 0, q_2 \to 1)$ the automaton has to choose $g_1 \bar{g}_2$ in order to avoid one of the losing dead-end states. If we solve this game, we conclude that only the following five states are winning: $(q_0 \to 0)$, (t_1), (t_2), $(q_0 \to 0, q_2 \to 1)$, and $(q_0 \to 0, q_1 \to 1)$. The bold arrows in Fig. 10 show a winning strategy, which corresponds to the system shown in Fig. 9. The system sends by default a grant to Client 2. If it receives a request from Client 1, it responds to it in the next step. If it receives a request from Client 2 while sending a grant to Client 1, it will respond to it in the next step. If no request is outstanding it moves back to the default behavior (i.e., sending a grant to Client 2).

Step 4: System Construction

Once a winning strategy is found, we can construct the desired system following Step 5 of the classical approach.

27.3.5.2 Approaches for Fragments of LTL

We will now consider another approach to making synthesis more efficient by considering specification with restricted expressiveness.

The four simplest LTL fragments are (i) invariants ($\Box p$), (ii) reachability properties ($\Diamond p$), (iii) recurrence properties ($\Box \Diamond p$), and (iv) persistence properties ($\Diamond \Box p$). These fragments can be translated directly into the corresponding synthesis games: invariants translate into safety games, reachability properties into reachability games, recurrence properties into Büchi games, and persistence properties into co-Büchi games. Each of these fragments by itself is not expressive enough to specify a complete system. However, they are very useful in the context of controller synthesis [147, 148]. For example, given a system with deadlocks, we can ask for a controlled system that is deadlock-free.

Alur and La Torre [10] provide a comprehensive study of generators for deterministic automata and complexity analysis of various LTL fragments. Here, we will focus on a recent approach by Piterman, Pnueli, and Sa'ar [140] for LTL formulas in the Generalized Reactivity-1 (GR(1)) fragment, because this fragment lends itself to an efficient symbolic implementation.

Specifications in GR-(1) are of the form

$$\mathsf{env}_1 \wedge \cdots \wedge \mathsf{env}_n \rightarrow \mathsf{sys}_1 \wedge \cdots \wedge \mathsf{sys}_m,$$

where every sub-formula env_i and sys_i can be represented by a deterministic Büchi automaton.[3] The intuition is that $\mathsf{env}_1, \ldots, \mathsf{env}_n$ are formulas describing assumptions on the environment and $\mathsf{sys}_1, \ldots, \mathsf{sys}_m$ specify the desired behavior of the system if all the environment assumptions are satisfied.

The approach proceeds as follows: first, every sub-formula env_i (sys_i) is translated into a deterministic Büchi automaton. Each automaton is represented symbolically by (i) an initial predicate, (ii) a transition predicate, and (iii) a predicate describing the Büchi states. The initial and Büchi predicate refer to the atomic propositions and the set of state variables. As usual, the transition predicate may refer to the current and next values of the atomic propositions and the state variables. Then, the initial and transition predicates obtained from the different sub-formulas are conjoined to make a single initial predicate and a single transition system. On this transition system, we define the following acceptance condition using the set of Büchi predicates $\phi_1^e, \ldots, \phi_n^e$ obtained from environment assumptions and the predicates $\phi_1^s, \ldots, \phi_m^s$ obtained from the system guarantees. A path ρ through the transition system is accepting iff all system predicates ϕ_i^s are true infinitely often along the path or if some environment predicate ϕ_i^e is true only finitely often along the path. This is a generalized Streett condition with a single Streett pair (see Sect. 27.2.4). Finally, this transition system is transformed into a game by splitting the transition predicate into two parts: one part that modifies the input variables and one part that modifies the output variables. In this way, we obtain a symbolic representation of a generalized Streett-1 game, which can be solved symbolically using a triply nested fix-point (see [140] and Sect. 27.2.4).

There are several further approaches whose strength is based on a decomposition of the specification (according to the top-level Boolean structure, for instance) together with appropriate handling of the parts. Notable examples are [82, 117, 132, 160]. The GR(1) synthesis algorithm was extended to specifications in the intersection of LTL and ACTL by Ehlers [81].

[3] One way to syntactically characterize such sub-formulas is to require them to be in the set $\mathrm{LTL}^{\mathrm{det}}$ [122], which is the set of formulas defined as follows:

$$\varphi ::= p \mid \varphi \wedge \varphi \mid \bigcirc \varphi \mid (p \wedge \varphi) \vee (\neg p \wedge \varphi) \mid (p \wedge \varphi)\,\mathsf{U}(\neg p \wedge \varphi) \mid (p \wedge \varphi)\,\mathsf{W}(\neg p \wedge \varphi),$$

where p is an arbitrary atomic proposition. Note that this set includes invariants ($\Box\, p$) and the formula $\neg p\,\mathsf{U}\, p$, which is equivalent to $\Diamond\, p$. In [140], the authors provide a different set of syntactic restrictions.

27.4 Related Topics

In this chapter we have considered perfect-information turn-based zero-sum games for synthesis from linear-time logical specifications. Perfect-information zero-sum games are used in several applications other than temporal logic synthesis.

In *controller synthesis*, we are given a nondeterministic system and we aim to restrict (*control*) the nondeterministic choices such that the controlled system satisfies the given specification. Synthesis from logical specifications and controller synthesis focus on different aspects of the synthesis problem. Research in the area of synthesis from logical specifications initially concentrated on developing new synthesis algorithms for more expressive logics, while controller synthesis focused on how to efficiently compute restrictions for a system composed of several sub-components. For an introduction to control theory of discrete-event systems we refer the reader to [31]. Discrete-event control theory has been used, for instance, to automatically avoid deadlocks in multi-threaded programs [179] or for synthesis of fault-tolerant systems [100].

Several approaches use controller synthesis to combine synthesis with imperative programming, thus avoiding the need to fully specify the system. Such approaches include program sketching [162, 163], program repair [11, 109], and synthesis of concurrent data structures [161] and synchronizations [172, 173]. It should be noted that some of these approaches use very different theory from what is presented in this chapter and that automatic programming has a much richer background than can be covered in this chapter [113, 114, 116]. See [21] for an overview of recent program synthesis techniques.

Other classes of games are also relevant in synthesis. While standard LTL synthesis reduces to perfect-information games, synthesis in the distributed setting reduces to multi-player partial-information games and distributed games [94, 118, 121, 144]. Distributed synthesis is undecidable in general but semi-decision procedures exist [156]. For the related case of parameterized systems, see [107].

The extension of synthesis to an assume-guarantee setting requires solving non-zero-sum games (namely secure equilibria) [55, 59]; and has been applied to protocol synthesis [69, 70]. Synthesis problems for resource constraints [32, 33], for performance guarantees [19], and for synthesis of robust systems [20] entail non-Boolean properties and reduce to games with quantitative objectives. Synthesis problems in probabilistic environments [12, 58, 152] or for synthesizing environment assumptions for synthesis reduce to stochastic games [57].

The games we have discussed are generalizations of finite automata. It is also possible to generalize other automata models, for instance to pushdown games [168, 177] or timed games. Timed games and controller synthesis for timed automata are discussed in Chap. 29 in this Handbook [23]. Finally, the close connection between games and verification is the subject of Chap. 26 in this Handbook [24].

References

1. Abadi, M., Lamport, L., Wolper, P.: Realizable and unrealizable specifications of reactive systems. In: Ausiello, G., Dezani-Ciancaglini, M., Della Rocca, S.R. (eds.) Intl. Colloquium on Automata, Languages and Programming (ICALP). LNCS, vol. 372, pp. 1–17. Springer, Heidelberg (1989)
2. de Alfaro, L., Faella, M., Henzinger, T.A., Majumdar, R., Stoelinga, M.: The element of surprise in timed games. In: Amadio, R.M., Lugiez, D. (eds.) Intl. Conf. on Concurrency Theory (CONCUR). LNCS, vol. 2761, pp. 144–158. Springer, Heidelberg (2003)
3. de Alfaro, L., Henzinger, T.A.: Interface automata. In: Tjoa, A.M., Gruhn, V. (eds.) Intl. Symp. on Foundations of Software Engineering (FSE), pp. 109–120. ACM, New York (2001)
4. de Alfaro, L., Henzinger, T.A.: Interface theories for component-based design. In: Henzinger, T.A., Kirsch, C.M. (eds.) Intl. Conf. on Embedded Software (EMSOFT). LNCS, vol. 2211, pp. 148–165. Springer, Heidelberg (2001)
5. de Alfaro, L., Henzinger, T.A., Kupferman, O.: Concurrent reachability games. Theor. Comput. Sci. **386**(3), 188–217 (2007)
6. de Alfaro, L., Henzinger, T.A., Mang, F.: Detecting errors before reaching them. In: Emerson, E.A., Sistla, A.P. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 1855, pp. 186–201. Springer, Heidelberg (2000)
7. de Alfaro, L., Majumdar, R.: Quantitative solution of omega-regular games. In: Vitter, J.S., Spirakis, P.G., Yannakakis, M. (eds.) Annual ACM Symposium on Theory of Computing (STOC), pp. 675–683. ACM, New York (2001)
8. Alur, R., Bodík, R., Dallal, E., Fisman, D., Garg, P., Juniwal, G., Kress-Gazit, H., Madhusudan, P., Martin, M.M.K., Raghothaman, M., Saha, S., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: Irlbeck, M., Peled, D.A., Pretschner, A. (eds.) Dependable Software Systems Engineering, pp. 1–25. IOS Press, Amsterdam (2015)
9. Alur, R., Henzinger, T.A., Kupferman, O.: Alternating-time temporal logic. J. ACM **49**, 672–713 (2002)
10. Alur, R., Torre, S.L.: Deterministic generators and games for LTL fragments. In: Symp. on Logic in Computer Science (LICS), pp. 291–302. IEEE, Piscataway (2001)
11. Antoniotti, M.: Synthesis and verification of discrete controllers for robotics and manufacturing devices with temporal logic and the Control-D system. Ph.D. thesis, New York University (1995)
12. Baier, C., Größer, M., Leucker, M., Bollig, B., Ciesinski, F.: Controller synthesis for probabilistic systems. In: IFIP Intl. Conf. on Theoretical Computer Science, pp. 493–506 (2004)
13. Beeri, C.: On the membership problem for functional and multivalued dependencies in relational databases. ACM Trans. Database Syst. **5**, 241–259 (1980)
14. Bertrand, N., Genest, B., Gimbert, H.: Qualitative determinacy and decidability of stochastic games with signals. In: Symp. on Logic in Computer Science (LICS), pp. 319–328. IEEE, Piscataway (2009)
15. Björklund, H., Sandberg, S., Vorobyov, S.: A discrete subexponential algorithms for parity games. In: Annual Symposium on Theoretical Aspects of Computer Science (STACS). LNCS, vol. 2607, pp. 663–674. Springer, Heidelberg (2003)
16. Bjorklund, H., Sandberg, S., Vorobyov, S.: A combinatorial strongly subexponential strategy improvement algorithm for mean payoff games. In: Intl. Symp. on Mathematical Foundations of Computer Science (MFCS). LNCS, vol. 3153, pp. 673–685. Springer, Heidelberg (2004)
17. Blass, A., Gurevich, Y., Nachmanson, L., Veanes, M.: Play to test. In: Intl. Conf. on Formal Approaches to Software Testing (FATES). LNCS, vol. 3997. Springer, Heidelberg (2005)
18. Bloem, R., Chatterjee, K., Greimel, K., Henzinger, T.A., Jobstmann, B.: Robustness in the presence of liveness. In: Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 6174, pp. 410–424. Springer, Heidelberg (2010)

19. Bloem, R., Chatterjee, K., Henzinger, T.A., Jobstmann, B.: Better quality in synthesis through quantitative objectives. In: Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 5643, pp. 140–156. Springer, Heidelberg (2009)
20. Bloem, R., Greimel, K., Henzinger, T.A., Jobstmann, B.: Synthesizing robust systems. In: Formal Methods in Computer Aided Design (FMCAD), pp. 85–92. IEEE, Piscataway (2009)
21. Bodík, R., Jobstmann, B.: Algorithmic program synthesis: introduction. Int. J. Softw. Tools Technol. Transf. **15**(5–6), 397–411 (2013)
22. Bouyer, P., Fahrenberg, U., Larsen, K.G., Markey, N., Srba, J.: Infinite runs in weighted timed automata with energy constraints. In: Intl. Conf. on Formal Modeling and Analysis of Timed Systems (FORMATS). LNCS, vol. 5215, pp. 33–47. Springer, Heidelberg (2008)
23. Bouyer, P., Fahrenberg, U., Larsen, K.G., Markey, N.: Ouaknine, J., Worrell, J., Verification of real-time systems. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
24. Bradfield, J., Walukiewicz, I.: The mu-calculus. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
25. Brázdil, T., Brozek, V., Chatterjee, K., Forejt, V., Kucera, A.: Two views on multiple mean-payoff objectives in Markov decision processes. In: Symp. on Logic in Computer Science (LICS), pp. 33–42. IEEE, Piscataway (2011)
26. Brázdil, T., Chatterjee, K., Forejt, V., Kucera, A.: Trading performance for stability in Markov decision processes. In: Symp. on Logic in Computer Science (LICS), pp. 331–340. IEEE, Piscataway (2013)
27. Brázdil, T., Chatterjee, K., Kucera, A., Novotný, P.: Efficient controller synthesis for consumption games with multiple resource types. In: Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 7358, pp. 23–38. Springer, Heidelberg (2012)
28. Brim, L., Chaloupka, J., Doyen, L., Gentilini, R., Raskin, J.F.: Faster algorithms for mean-payoff games. Form. Methods Syst. Des. **38**(2), 97–118 (2011)
29. Bryant, R.E.: Binary decision diagrams: an algorithmic basis for symbolic model checking. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
30. Büchi, J.R., Landweber, L.H.: Solving sequential conditions by finite-state strategies. Trans. Am. Math. Soc. **138**, 295–311 (1969)
31. Cassandras, C.G., Lafortune, S.: Introduction to Discrete Event Systems, 2nd edn. Springer, Heidelberg (2008)
32. Cerný, P., Chatterjee, K., Henzinger, T.A., Radhakrishna, A., Singh, R.: Quantitative synthesis for concurrent programs. In: Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 6806, pp. 243–259. Springer, Heidelberg (2011)
33. Chakrabarti, A., de Alfaro, L., Henzinger, T.A., Stoelinga, M.: Resource interfaces. In: Intl. Conf. on Embedded Software (EMSOFT). LNCS, vol. 2855, pp. 117–133. Springer, Heidelberg (2003)
34. Chatterjee, K.: Stochastic ω-regular games. Ph.D. thesis, UC Berkeley (2007)
35. Chatterjee, K.: The complexity of stochastic Müller games. Inf. Comput. **211**, 29–48 (2012)
36. Chatterjee, K., de Alfaro, L., Henzinger, T.A.: The complexity of stochastic Rabin and Streett games. In: Intl. Colloquium on Automata, Languages and Programming (ICALP). LNCS, vol. 3580, pp. 878–890. Springer, Heidelberg (2005)
37. Chatterjee, K., de Alfaro, L., Henzinger, T.A.: The complexity of quantitative concurrent parity games. In: ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 678–687. ACM/SIAM, New York/Philadelphia (2006)
38. Chatterjee, K., de Alfaro, L., Henzinger, T.A.: Strategy improvement in concurrent reachability games. In: Intl. Conf. on Quantitative Evaluation of Systems (QEST), pp. 291–300. IEEE, Piscataway (2006)
39. Chatterjee, K., de Alfaro, L., Henzinger, T.A.: Qualitative concurrent parity games. ACM Trans. Comput. Log. **12**(4), 28 (2011)

40. Chatterjee, K., Chmelik, M., Tracol, M.: What is decidable about partially observable Markov decision processes with omega-regular objectives. In: Annual Conf. on Computer Science Logic (CSL). LIPIcs, vol. 23 (2013)
41. Chatterjee, K., Doyen, L.: The complexity of partial-observation parity games. In: Intl. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR). LNCS, vol. 6397, pp. 1–14. Springer, Heidelberg (2010)
42. Chatterjee, K., Doyen, L.: Energy parity games. Theor. Comput. Sci. **458**, 49–60 (2012)
43. Chatterjee, K., Doyen, L.: Partial-observation stochastic games: how to win when belief fails. In: Symp. on Logic in Computer Science (LICS), pp. 175–184. IEEE, Piscataway (2012)
44. Chatterjee, K., Doyen, L., Gimbert, H., Henzinger, T.A.: Randomness for free. In: Intl. Symp. on Mathematical Foundations of Computer Science (MFCS). LNCS, vol. 6281, pp. 246–257. Springer, Heidelberg (2010)
45. Chatterjee, K., Doyen, L., Henzinger, T., Raskin, J.F.: Generalized mean-payoff and energy games. In: Annual Conf. on Foundations of Software Technology and Theoretical Computer Science (FSTTCS). LIPIcs, vol. 8, pp. 505–516. Schloss Dagstuhl—LZI, Dagstuhl (2010)
46. Chatterjee, K., Doyen, L., Henzinger, T.A.: Qualitative analysis of partially-observable Markov decision processes. In: Intl. Symp. on Mathematical Foundations of Computer Science (MFCS). LNCS, vol. 6281, pp. 258–269. Springer, Heidelberg (2010)
47. Chatterjee, K., Doyen, L., Henzinger, T.A.: A survey of partial-observation stochastic parity games. Form. Methods Syst. Des. **43**(2), 268–284 (2013)
48. Chatterjee, K., Doyen, L., Henzinger, T.A., Raskin, J.: Algorithms for omega-regular games with imperfect information. In: Annual Conf. on Computer Science Logic (CSL). LNCS, vol. 4207, pp. 287–302. Springer, Heidelberg (2006)
49. Chatterjee, K., Doyen, L., Nain, S., Vardi, M.Y.: The complexity of partial-observation stochastic parity games with finite-memory strategies. In: Intl. Conf. on Foundations of Software Science and Computational Structures (FoSSaCS). LNCS, vol. 8412. Springer, Heidelberg (2014)
50. Chatterjee, K., Forejt, V., Wojtczak, D.: Multi-objective discounted reward verification in graphs and MDPs. In: Intl. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR). LNCS, vol. 8312. Springer, Heidelberg (2013)
51. Chatterjee, K., Goyal, P., Ibsen-Jensen, R., Pavlogiannis, A.: Faster algorithms for algebraic path properties in recursive state machines with constant treewidth. In: Symp. on Principles of Programming Languages (POPL). ACM, New York (2015)
52. Chatterjee, K., Henzinger, M.: An $O(n^2)$ time algorithm for alternating Büchi games. In: ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 1386–1399 (2012)
53. Chatterjee, K., Henzinger, M.: Efficient and dynamic algorithms for alternating Büchi games and maximal end-component decomposition. J. ACM **61**(3), 15:1–15:40 (2014)
54. Chatterjee, K., Henzinger, T.A.: Semiperfect-information games. In: Annual Conf. on Foundations of Software Technology and Theoretical Computer Science (FSTTCS). LNCS, vol. 3821, pp. 1–18. Springer, Heidelberg (2005)
55. Chatterjee, K., Henzinger, T.A.: Assume-guarantee synthesis. In: Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 4424, pp. 261–275. Springer, Heidelberg (2007)
56. Chatterjee, K., Henzinger, T.A.: A survey of stochastic omega-regular games. J. Comput. Syst. Sci. **78**(2), 394–413 (2012)
57. Chatterjee, K., Henzinger, T.A., Jobstmann, B.: Environment assumptions for synthesis. In: Intl. Conf. on Concurrency Theory (CONCUR). CONCUR, vol. 2008, pp. 147–161. Springer, Heidelberg (2008)
58. Chatterjee, K., Henzinger, T.A., Jobstmann, B., Singh, R.: Measuring and synthesizing systems in probabilistic environments. In: Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 6174, pp. 380–395. Springer, Heidelberg (2010)
59. Chatterjee, K., Henzinger, T.A., Jurdziński, M.: Games with secure equilibria. In: Symp. on Logic in Computer Science (LICS), pp. 160–169. IEEE, Piscataway (2004)

60. Chatterjee, K., Henzinger, T.A., Jurdzinski, M.: Mean-payoff parity games. In: Symp. on Logic in Computer Science (LICS), pp. 178–187 (2005)
61. Chatterjee, K., Henzinger, T.A., Piterman, N.: Algorithms for Büchi games. In: Workshop on Games in Design and Verification (GDV) (2006)
62. Chatterjee, K., Henzinger, T.A., Piterman, N.: Generalized parity games. In: Intl. Conf. on Foundations of Software Science and Computational Structures (FoSSaCS). LNCS, vol. 4423, pp. 153–167. Springer, Heidelberg (2007)
63. Chatterjee, K., Henzinger, T.A., Piterman, N.: Strategy logic. Inf. Comput. **208**(6), 677–693 (2010)
64. Chatterjee, K., Jurdziński, M., Henzinger, T.A.: Simple stochastic parity games. In: Annual Conf. on Computer Science Logic (CSL). LNCS, vol. 2803, pp. 100–113. Springer, Heidelberg (2003)
65. Chatterjee, K., Jurdziński, M., Henzinger, T.A.: Quantitative stochastic parity games. In: ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 121–130. SIAM, Philadelphia (2004)
66. Chatterjee, K., Majumdar, R., Henzinger, T.A.: Markov decision processes with multiple objectives. In: Annual Symposium on Theoretical Aspects of Computer Science (STACS). LNCS, vol. 3884, pp. 325–336. Springer, Heidelberg (2006)
67. Chatterjee, K., Majumdar, R., Henzinger, T.A.: Stochastic limit-average games are in EXPTIME. Int. J. Game Theory **37**(2), 219–234 (2008)
68. Chatterjee, K., Pavlogiannis, A., Velner, Y.: Quantitative interprocedural analysis. In: Symp. on Principles of Programming Languages (POPL). ACM, New York (2015)
69. Chatterjee, K., Raman, V.: Synthesizing protocols for digital contract signing. In: Intl. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI). LNCS, vol. 7148, pp. 152–168. Springer, Heidelberg (2012)
70. Chatterjee, K., Raman, V.: Assume-guarantee synthesis for digital contract signing. Form. Asp. Comput. **26**(4), 825–859 (2014)
71. Chatterjee, K., Randour, M., Raskin, J.F.: Strategy synthesis for multi-dimensional quantitative objectives. In: Intl. Conf. on Concurrency Theory (CONCUR). LNCS, vol. 7454, pp. 115–131. Springer, Heidelberg (2012)
72. Chatterjee, K., Velner, Y.: Mean-payoff pushdown games. In: Symp. on Logic in Computer Science (LICS), pp. 195–204. IEEE, Piscataway (2012)
73. Chatterjee, K., Velner, Y.: Hyperplane separation technique for multidimensional mean-payoff games. In: Intl. Conf. on Concurrency Theory (CONCUR). LNCS, vol. 8052, pp. 500–515. Springer, Heidelberg (2013)
74. Chen, T., Forejt, V., Kwiatkowska, M.Z., Simaitis, A., Wiltsche, C.: On stochastic games with multiple objectives. In: Intl. Symp. on Mathematical Foundations of Computer Science (MFCS). LNCS, vol. 8087, pp. 266–277. Springer, Heidelberg (2013)
75. Church, A.: Logic, arithmetic, and automata. In: Proceedings of the International Congress of Mathematicians, pp. 23–35. Institut Mittag-Leffler, Djursholm (1962)
76. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching-time temporal logic. In: Kozen, D. (ed.) Logic of Programs. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1981)
77. Condon, A.: The complexity of stochastic games. Inf. Comput. **96**(2), 203–224 (1992)
78. Condon, A.: On algorithms for simple stochastic games. In: Advances in Computational Complexity Theory. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 13, pp. 51–73. AMS, Providence (1993)
79. Dill, D.L.: Trace theory for automatic hierarchical verification of speed-independent circuits. Ph.D. thesis, ACM Distinguished Dissertation Series. MIT Press (1989)
80. Dziembowski, S., Jurdziński, M., Walukiewicz, I.: How much memory is needed to win infinite games? In: Symp. on Logic in Computer Science (LICS), pp. 99–110. IEEE, Piscataway (1997)
81. Ehlers, R.: ACTL ∩ LTL synthesis. In: Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 7358, pp. 39–54. Springer, Heidelberg (2012)

82. Ehlers, R.: Symbolic bounded synthesis. Form. Methods Syst. Des. **40**(2), 232–262 (2012)
83. Ehrenfeucht, A., Mycielski, J.: Positional strategies for mean payoff games. Int. J. Game Theory **8**(2), 109–113 (1979)
84. Emerson, E.A., Clarke, E.M.: Using branching time temporal logic to synthesize synchronization skeletons. Sci. Comput. Program. **2**(3), 241–266 (1982)
85. Emerson, E.A., Jutla, C.S.: The complexity of tree automata and logics of programs. In: Annual Symp. on Foundations of Computer Science (FOCS), pp. 328–337. IEEE, Piscataway (1988)
86. Emerson, E.A., Jutla, C.S.: Tree automata, mu-calculus and determinacy. In: Annual Symp. on Foundations of Computer Science (FOCS), pp. 368–377. IEEE, Piscataway (1991)
87. Etessami, K., Kwiatkowska, M.Z., Vardi, M.Y., Yannakakis, M.: Multi-objective model checking of Markov decision processes. Log. Methods Comput. Sci. **4**(4) (2008)
88. Etessami, K., Wilke, T., Schuller, R.A.: Fair simulation relations, parity games, and state space reduction for Büchi automata. SIAM J. Comput. **34**(5), 1159–1175 (2005)
89. Etessami, K., Yannakakis, M.: Recursive Markov decision processes and recursive stochastic games. In: Intl. Colloquium on Automata, Languages and Programming (ICALP). LNCS, vol. 3580, pp. 891–903. Springer, Heidelberg (2005)
90. Etessami, K., Yannakakis, M.: Recursive concurrent stochastic games. In: Intl. Colloquium on Automata, Languages and Programming (ICALP). LNCS, vol. 4052, pp. 324–335. Springer, Heidelberg (2006)
91. Etessami, K., Yannakakis, M.: On the complexity of Nash equilibria and other fixed points. SIAM J. Comput. **39**(6), 2531–2597 (2010)
92. Filar, J., Vrieze, K.: Competitive Markov Decision Processes. Springer, Heidelberg (1997)
93. Filiot, E., Jin, N., Raskin, J.F.: An antichain algorithm for LTL realizability. In: Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 5643, pp. 263–277. Springer, Heidelberg (2009)
94. Finkbeiner, B., Schewe, S.: Uniform distributed synthesis. In: Symp. on Logic in Computer Science (LICS), pp. 321–330. IEEE, Piscataway (2005)
95. Finkbeiner, B., Schewe, S.: Coordination logic. In: Annual Conf. on Computer Science Logic (CSL). LNCS, vol. 6247, pp. 305–319. Springer, Heidelberg (2010)
96. Forejt, V., Kwiatkowska, M.Z., Norman, G., Parker, D., Qu, H.: Quantitative multi-objective verification for probabilistic systems. In: Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 6065, pp. 112–127. Springer, Heidelberg (2011)
97. Friedmann, O.: An exponential lower bound for the parity game strategy improvement algorithm as we know it. In: Symp. on Logic in Computer Science (LICS), pp. 145–156. IEEE, Piscataway (2009)
98. Friedmann, O.: Exponential lower bounds for solving infinitary payoff games and linear programs. Ph.D. thesis, University of Munich (2011)
99. Gastin, P., Oddoux, D.: Fast LTL to Büchi automata translation. In: Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 2102, pp. 53–65. Springer, Heidelberg (2001)
100. Girault, A., Rutten, É.: Automating the addition of fault tolerance with discrete controller synthesis. Form. Methods Syst. Des. **35**(2), 190–225 (2009)
101. Gurevich, Y., Harrington, L.: Trees, automata, and games. In: Annual ACM Symposium on Theory of Computing (STOC), pp. 60–65. ACM, New York (1982)
102. Hansen, K.A., Koucký, M., Lauritzen, N., Miltersen, P.B., Tsigaridas, E.P.: Exact algorithms for solving stochastic games: extended abstract. In: Annual ACM Symposium on Theory of Computing (STOC), pp. 205–214 (2011)
103. Henzinger, T.A., Krishnan, S., Kupferman, O., Mang, F.: Synthesis of uninitialized systems. In: Intl. Colloquium on Automata, Languages and Programming (ICALP). LNCS, vol. 2380, pp. 644–656. Springer, Heidelberg (2002)
104. Henzinger, T.A., Kupferman, O., Rajamani, S.: Fair simulation. Inf. Comput. **173**, 64–81 (2002)

105. Horn, F.: Streett games on finite graphs. In: Workshop on Games in Design and Verification (GDV) (2005)
106. Immerman, N.: Number of quantifiers is better than number of tape cells. J. Comput. Syst. Sci. **22**, 384–406 (1981)
107. Jacobs, S., Bloem, R.: Parameterized synthesis. In: Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 7214, pp. 362–376. Springer, Heidelberg (2012)
108. Jobstmann, B., Bloem, R.: Optimizations for LTL synthesis. In: Formal Methods in Computer Aided Design (FMCAD), pp. 117–124. IEEE, Piscataway (2006)
109. Jobstmann, B., Staber, S., Griesmayer, A., Bloem, R.: Finding and fixing faults. J. Comput. Syst. Sci. **78**(2), 441–460 (2012)
110. Jurdziński, M.: Deciding the winner in parity games is in UP ∩ co-UP. Inf. Process. Lett. **68**(3), 119–124 (1998)
111. Jurdziński, M.: Small progress measures for solving parity games. In: Annual Symposium on Theoretical Aspects of Computer Science (STACS). LNCS, vol. 1770, pp. 290–301. Springer, Heidelberg (2000)
112. Jurdziński, M., Paterson, M., Zwick, U.: A deterministic subexponential algorithm for solving parity games. In: ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 117–123. ACM/SIAM, New York/Philadelphia (2006)
113. Katz, G., Peled, D.: Synthesizing solutions to the leader election problem using model checking and genetic programming. In: Intl. Haifa Verification Conference (HVC). LNCS, vol. 6405, pp. 117–132. Springer, Heidelberg (2009)
114. Kitzelmann, E.: Inductive programming: a survey of program synthesis techniques. In: Intl. Workshop on Approaches and Applications of Inductive Programming (AAIP), pp. 50–73 (2009)
115. Klein, U., Piterman, N., Pnueli, A.: Effective synthesis of asynchronous systems from GR(1) specifications. In: Intl. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI). LNCS, vol. 7148, pp. 283–298. Springer, Heidelberg (2012)
116. Kuncak, V., Mayer, M., Piskac, R., Suter, P.: Complete functional synthesis. In: Conf. on Programming Language Design and Implementation (PLDI), pp. 316–329. ACM, New York (2010)
117. Kupferman, O., Piterman, N., Vardi, M.Y.: Safraless compositional synthesis. In: Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 4144, pp. 31–44. Springer, Heidelberg (2006)
118. Kupferman, O., Vardi, M.: Synthesis with incomplete information. In: 2nd International Conference on Temporal Logic, Manchester, pp. 91–106 (1997)
119. Kupferman, O., Vardi, M.: Weak alternating automata and tree automata emptiness. In: Annual ACM Symposium on Theory of Computing (STOC), pp. 224–233. ACM, New York (1998)
120. Kupferman, O., Vardi, M.Y.: Safraless decision procedures. In: Annual Symp. on Foundations of Computer Science (FOCS), pp. 531–542. IEEE, Piscataway (2005)
121. Madusudan, P.: Control and synthesis of open reactive systems. Ph.D. thesis, Theoretical Computer Science Group, Institute of Mathematical Sciences, University of Madras (2001)
122. Maidl, M.: The common fragment of CTL and LTL. In: Annual Symp. on Foundations of Computer Science (FOCS), pp. 643–652. IEEE, Piscataway (2000)
123. Maler, O., Pnueli, A., Sifakis, J.: On the synthesis of discrete controllers for timed systems. In: Annual Symposium on Theoretical Aspects of Computer Science (STACS). LNCS, vol. 900, pp. 229–242. Springer, Heidelberg (1995)
124. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems: Specification. Springer, Heidelberg (1992)
125. Manna, Z., Pnueli, A.: Temporal Verification of Reactive Systems. Springer, Heidelberg (1995)
126. Manna, Z., Wolper, P.: Synthesis of communicating processes from temporal logic specifications. Trans. Program. Lang. Syst. **6**(1), 68–93 (1984)

127. Martin, D.: The determinacy of Blackwell games. J. Symb. Log. **63**(4), 1565–1581 (1998)
128. McNaughton, R.: Infinite games played on finite graphs. Ann. Pure Appl. Log. **65**, 149–184 (1993)
129. Miyano, S., Hayashi, T.: Alternating finite automata on omega-words. Theor. Comput. Sci. **32**, 321–330 (1984)
130. Mogavero, F., Murano, A., Sauro, L.: On the boundary of behavioral strategies. In: Symp. on Logic in Computer Science (LICS), pp. 263–272 (2013)
131. Mogavero, F., Murano, A., Vardi, M.Y.: Reasoning about strategies. In: Annual Conf. on Foundations of Software Technology and Theoretical Computer Science (FSTTCS). LIPIcs, vol. 8, pp. 133–144 (2010)
132. Morgenstern, A.: Symbolic controller synthesis for LTL specifications. Ph.D. thesis, TU Kaiserslautern (2010)
133. Mostowski, A.: Regular expressions for infinite trees and a standard form of automata. In: Symposium on Computation Theory. LNCS, vol. 208, pp. 157–168. Springer, Heidelberg (1984)
134. Muller, D.E., Schupp, P.E.: Alternating automata on infinite objects, determinacy and Rabin's theorem. In: Automata on Infinite Words. LNCS, vol. 192, pp. 100–107. Springer, Heidelberg (1984)
135. Nain, S., Vardi, M.Y.: Solving partial-information stochastic parity games. In: Symp. on Logic in Computer Science (LICS), pp. 341–348. IEEE, Piscataway (2013)
136. Papadimitriou, C.H., Tsitsiklis, J.N.: The complexity of Markov decision processes. Math. Oper. Res. **12**, 441–450 (1987)
137. Piterman, N.: From nondeterministic Büchi and Streett automata to deterministic parity automata. In: Symp. on Logic in Computer Science (LICS), pp. 255–264. IEEE, Seattle (2006)
138. Piterman, N., Pnueli, A.: Faster solution of Rabin and Streett games. In: Symp. on Logic in Computer Science (LICS), pp. 275–284. IEEE, Piscataway (2006)
139. Piterman, N., Pnueli, A.: Temporal logic and fair discrete systems. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
140. Piterman, N., Pnueli, A., Sa´ar, Y.: Synthesis of reactive(1) designs. In: Intl. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI). LNCS, vol. 3855, pp. 364–380. Springer, Heidelberg (2006)
141. Pnueli, A.: The temporal logic of programs. In: Annual Symp. on Foundations of Computer Science (FOCS), pp. 46–57. IEEE, Piscataway (1977)
142. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Symp. on Principles of Programming Languages (POPL), pp. 179–190. ACM, New York (1989)
143. Pnueli, A., Rosner, R.: On the synthesis of an asynchronous reactive module. In: Intl. Colloquium on Automata, Languages and Programming (ICALP). LNCS, vol. 372, pp. 652–671. Springer, Heidelberg (1989)
144. Pnueli, A., Rosner, R.: Distributed reactive systems are hard to synthesize. In: Annual Symp. on Foundations of Computer Science (FOCS), pp. 746–757. IEEE, Piscataway (1990)
145. Puchala, B.: Asynchronous omega-regular games with partial information. In: Intl. Symp. on Mathematical Foundations of Computer Science (MFCS). LNCS, vol. 6281, pp. 592–603. Springer, Heidelberg (2010)
146. Rabin, M.O.: Decidability of second-order theories and automata on infinite trees. Trans. Am. Math. Soc. **141**, 1–35 (1969)
147. Ramadge, P.J., Wonham, W.M.: Supervisory control of a class of discrete event processes. SIAM J. Control Optim. **25**, 206–230 (1987)
148. Ramadge, P.J.G., Wonham, W.M.: The control of discrete event systems. Proc. IEEE **77**, 81–98 (1989)
149. Reif, J.H.: Universal games of incomplete information. In: Annual ACM Symposium on Theory of Computing (STOC), pp. 288–308. ACM, New York (1979)
150. Rosner, R.: Modular synthesis of reactive systems. Ph.D. thesis, Weizmann Institute of Science (1992)

151. Safra, S.: On the complexity of ω-automata. In: Annual Symp. on Foundations of Computer Science (FOCS), pp. 319–327. IEEE, Piscataway (1988)
152. Schewe, S.: Synthesis for probabilistic environments. In: Intl. Symp. Automated Technology for Verification and Analysis (ATVA). LNCS, vol. 4218, pp. 245–259. Springer, Heidelberg (2006)
153. Schewe, S.: Solving parity games in big steps. In: Annual Conf. on Foundations of Software Technology and Theoretical Computer Science (FSTTCS). LNCS, vol. 4855, pp. 449–460. Springer, Heidelberg (2007)
154. Schewe, S.: An optimal strategy improvement algorithm for solving parity and payoff games. In: Annual Conf. on Computer Science Logic (CSL). LNCS, vol. 5213, pp. 369–384. Springer, Heidelberg (2008)
155. Schewe, S., Finkbeiner, B.: Synthesis of asynchronous systems. In: Intl. Sym. on Logic-Based Program Synthesis and Transformation, 16th International (LOPSTR). LNCS, vol. 4407, pp. 127–142. Springer, Heidelberg (2006)
156. Schewe, S., Finkbeiner, B.: Bounded synthesis. In: Intl. Symp. Automated Technology for Verification and Analysis (ATVA). LNCS, vol. 4762, pp. 474–488. Springer, Heidelberg (2007)
157. Schneider, K.: Improving automata generation for linear temporal logic by considering the automaton hierarchy. In: Intl. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR). LNCS, vol. 2250, pp. 39–54. Springer, Heidelberg (2001)
158. Sebastiani, R., Tonetta, S.: "More deterministic" vs. "smaller" Büchi automata for efficient LTL model checking. In: Correct Hardware Design and Verification Methods (CHARME). LNCS, vol. 2860, pp. 126–140. Springer, Heidelberg (2003)
159. Shapley, L.: Stochastic games. Proc. Natl. Acad. Sci. USA **39**, 1095–1100 (1953)
160. Sohail, S., Somenzi, F.: Safety first: a two-stage algorithm for LTL games. In: Formal Methods in Computer Aided Design (FMCAD), pp. 77–84. IEEE, Piscataway (2009)
161. Solar-Lezama, A., Jones, C.G., Bodík, R.: Sketching concurrent data structures. In: Conf. on Programming Language Design and Implementation (PLDI), pp. 136–148. ACM, New York (2008)
162. Solar-Lezama, A., Rabbah, R.M., Bodík, R., Ebcioglu, K.: Programming by sketching for bit-streaming programs. In: Conf. on Programming Language Design and Implementation (PLDI), pp. 281–294. ACM, New York (2005)
163. Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S.A., Saraswat, V.A.: Combinatorial sketching for finite programs. In: Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 404–415. ACM, New York (2006)
164. Somenzi, F.: Colorado University Decision Diagram Package (1998). http://vlsi.colorado.edu/~fabio/CUDD/
165. Somenzi, F., Bloem, R.: Efficient Büchi automata from LTL formulae. In: Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 1855, pp. 248–263. Springer, Heidelberg (2000)
166. Stockmeyer, L.: The complexity of decision problems in automata theory and logic. Ph.D. thesis, Massachusetts Institute of Technology (1974)
167. Thomas, W.: Automata on infinite objects. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, vol. B, pp. 133–191. Elsevier, Amsterdam (1990)
168. Thomas, W.: On the synthesis of strategies in infinite games. In: Annual Symposium on Theoretical Aspects of Computer Science (STACS). LNCS, vol. 900, pp. 1–13. Springer, Heidelberg (1995)
169. Thomas, W.: Languages, automata, and logic. In: Rozenberg, G., Salomaa, A. (eds.) Handbook of Formal Languages, vol. 3, pp. 389–455. Springer, Heidelberg (1997). Beyond Words, Chap. 7
170. Vardi, M.Y.: An automata-theoretic approach to fair realizability and synthesis. In: Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 939, pp. 267–278. Springer, Heidelberg (1995)

171. Vardi, M.Y., Wolper, P.: Reasoning about infinite computations. Inf. Comput. **115**(1), 1–37 (1994)
172. Vechev, M.T., Yahav, E., Yorsh, G.: Inferring synchronization under limited observability. In: Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 5505, pp. 139–154. Springer, Heidelberg (2009)
173. Vechev, M.T., Yahav, E., Yorsh, G.: Abstraction-guided synthesis of synchronization. In: Symp. on Principles of Programming Languages (POPL), pp. 327–338. ACM, New York (2010)
174. Velner, Y., Rabinovich, A.: Church synthesis problem for noisy input. In: Intl. Conf. on Foundations of Software Science and Computational Structures (FoSSaCS). LNCS, vol. 6604, pp. 275–289. Springer, Heidelberg (2011)
175. Vöge, J., Jurdziński, M.: A discrete strategy improvement algorithm for solving parity games. In: Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 1855, pp. 202–215. Springer, Heidelberg (2000)
176. Walukiewicz, I.: Pushdown processes: games and model checking. In: Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 1102, pp. 62–74. Springer, Heidelberg (1996)
177. Walukiewicz, I.: Pushdown processes: games and model-checking. Inf. Comput. **164**(2), 234–263 (2001)
178. Walukiewicz, I.: A landscape with games in the background. In: Symp. on Logic in Computer Science (LICS), pp. 356–366. IEEE, Piscataway (2004)
179. Wang, Y., Lafortune, S., Kelly, T., Kudlur, M., Mahlke, S.A.: The theory of deadlock avoidance via discrete control. In: Symp. on Principles of Programming Languages (POPL), pp. 252–263. ACM, New York (2009)
180. Zielonka, W.: Infinite games on finitely coloured graphs with applications to automata on infinite trees. Theor. Comput. Sci. **200**(1–2), 135–183 (1998)
181. Zwick, U., Paterson, M.: The complexity of mean payoff games on graphs. Theor. Comput. Sci. **158**, 343–359 (1996)

Chapter 28
Model Checking Probabilistic Systems

Christel Baier, Luca de Alfaro, Vojtěch Forejt, and Marta Kwiatkowska

Abstract The model-checking approach was originally formulated for verifying qualitative properties of systems, for example safety and liveness (see Chap. 2), and subsequently extended to also handle quantitative features, such as real time (see Chap. 29), continuous flows (see Chap. 30), as well as stochastic phenomena, where system evolution is governed by a given probability distribution. Probabilistic model checking aims to establish the correctness of probabilistic system models against quantitative probabilistic specifications, such as those capable of expressing, for example, the probability of an unsafe event occurring, expected time to termination, or expected power consumption in the start-up phase. In this chapter, we present the foundations of probabilistic model checking, focusing on finite-state Markov decision processes as models and quantitative properties expressed in probabilistic temporal logic. Markov decision processes can be thought of as a probabilistic variant of labelled transition systems in the following sense: transitions are labelled with actions, which can be chosen nondeterministically, and successor states for the chosen action are specified by means of discrete probabilistic distributions, thus specifying the probability of transiting to each successor state. To reason about expectations, we additionally annotate Markov decision processes with quantitative costs, which are incurred upon taking the selected action from a given state. Quantitative properties are expressed as formulas of the probabilistic computation tree logic (PCTL) or using linear temporal logic (LTL). We summarise the main model-checking algorithms for both PCTL and LTL, and illustrate their working through examples. The chapter ends with a brief overview of extensions to more expressive models and temporal logics, existing probabilistic model-checking tool support, and main application domains.

C. Baier
Technische Universität Dresden, Dresden, Germany

L. de Alfaro
University of California, Santa Cruz, Santa Cruz, CA, USA

V. Forejt · M. Kwiatkowska (✉)
University of Oxford, Oxford, UK
e-mail: marta.kwiatkowska@cs.ox.ac.uk

E.M. Clarke et al. (eds.), *Handbook of Model Checking*,
DOI 10.1007/978-3-319-10575-8_28

28.1 Introduction

Markovian stochastic models, i.e., state-transition graphs annotated with probabilities to model and reason about stochastic phenomena, are central to many applications. Traditionally, purely stochastic models such as Markov chains [96] have been applied in, for example, queueing theory, performance evaluation, and the modelling of telecommunication systems and networks [13, 21, 61], but they are also widely used in other contexts. Dependability properties such as reliability and availability are expressed probabilistically. In systems biology, for example, stochastic models can be used to reason about biological populations and the evolution of concentrations of molecules in biological signalling networks [62]. Probabilistic models with nondeterminism, for example Markov decision processes (abbreviated as MDPs) [99], which are the main focus of this chapter, are central to the modelling of distributed coordination protocols that use randomization for medium access control for wireless networks [85], breaking the symmetry in leader election algorithms [68], or modelling security, anonymity and privacy protocols [90], among many examples. MDPs are also widely used in operations research, economics, robotics, and related disciplines that crucially rely on the concept of decision making so as to choose the next action to optimize a certain goal function. Another application of MDPs is modelling distributed systems that operate with unreliable components. For instance, for systems with communication channels that might corrupt or lose messages, or interact with sensors that deliver wrong values in certain cases, probability distributions can be used to specify the frequency of faulty behaviour. Considering stochastic models more generally, further examples are—ranking algorithms in search engines for the Internet, the analysis of soccer or baseball matches, reasoning about the stochastic growth of waves of influenza or the population dynamics of other pathogenic germs, speech recognition, and signature recognition via biometric identification features. We give a brief overview of related models at the end of this chapter.

28.1.1 Temporal Logics for Specifying Probabilistic Properties

Probabilistic temporal logics arise as generalisations of established temporal logics such as computation tree logic (CTL) and linear temporal logic (LTL). Probabilistic computation tree logic (PCTL) [15, 17, 59] is a probabilistic variant of CTL that replaces the usual path quantifiers, with which one can reason about all or some paths satisfying a certain condition, with operators instead imposing quantitative constraints on the proportion of paths that satisfy this condition. More specifically, PCTL provides a probabilistic operator whose role is to specify lower or upper probability bounds for reachability properties, in the sense of requiring that the probability of reaching a given set of states is above or below a given threshold value. The reachability properties can be constrained using the CTL path modality "until" U or its step-bounded variant $\mathsf{U}^{\leq k}$. For instance, using the probabilistic operator

one might formally establish the guarantee that a system failure will occur within the next 100 steps with probability 10^{-8} or less, or that a leader will eventually be elected almost surely, that is, with probability 1. Besides the probability operator, expected cost operators can also be defined, which allow for reasoning, for example, about the average cost to reach a certain set of target states, or the accumulated cost within the next k steps. The cost operators can, for instance, be used to assert that the expected energy consumption within the next 100 steps is less than a given threshold. For Markov decision processes decorated with costs, model checking reduces to the computation of the minimum or maximum probability/expectation values, over the possible resolutions of nondeterminism.

While PCTL is a branching-time logic and its formulas express properties that a state of a probabilistic model might or might not have, probabilistic systems can also be analysed using purely linear-time (path-based) formalisms such as LTL or automata over infinite words [11, 40, 97, 105, 106]. We will restrict our attention to the logic LTL in this chapter. Unlike PCTL, it does not admit path quantifiers, but it allows us to express more elaborate properties, because it is possible to combine temporal operators. One can then, for example, express a path property "whenever button 1 is pressed, the system will be operational until button 2 is pressed". Such a property would not be expressible in PCTL. Since the underlying model is probabilistic, after fixing an LTL formula we are interested in quantitatively reasoning about the proportion of the paths satisfying the specification, analogously to PCTL. For this purpose we introduce *LTL state properties*, which are given by an LTL formula and a probability bound, and are true in a state if the maximum probability of the formula being satisfied is lower than the bound given. The solution methods we present in this chapter also allow us to ask "quantitative" questions, i.e., to directly compute the maximum probability that a given LTL formula is satisfied.

The two ways of reasoning about properties of MDPs which we study in this chapter, i.e., PCTL and LTL state properties, offer different expressive power. Essentially, the properties one can capture are in the same spirit as those in the non-probabilistic variants, and hence we refer the reader to Chap. 2 for a comprehensive overview. As in the non-probabilistic case, the properties expressed using LTL are perhaps easier to obtain from requirements expressed in natural language than PCTL formulas, but PCTL admits better complexity of model-checking algorithms, which are also easier to implement. We note that the two logics, PCTL and LTL, can be combined into a logic PCTL*.

In this chapter we will present the model-checking approach for Markov decision processes (MDPs) [4, 87, 99], which for the purposes of the model-checking algorithms discussed here are equivalent to probabilistic automata due to Segala [101, 102]. MDPs are of fundamental importance in probabilistic verification, since they not only serve as a natural representation of many real-world applications, for example distributed network protocols, but are also key to formulating abstractions for more complex models which incorporate dense real time and probability, such as continuous-time Markovian models and probabilistic variants of timed automata. Both PCTL and LTL can be used for reasoning about qualitative and quantitative properties of MDPs. Several variants of PCTL and LTL have been proposed for the

analysis of probabilistic models that rely on a dense time domain. These will be briefly addressed in Sect. 28.9.

28.1.2 Model-Checking Algorithms for Probabilistic Systems

For finite-state Markov decision processes, the quantitative analysis against PCTL or LTL specifications mainly relies on a combination of graph algorithms, automata-based constructions, and (numerical) algorithms for computing the minimum and maximum probabilities and expectation values. Compared to the non-probabilistic case, there is the additional difficulty of solving linear programs, and also the required graph algorithms are more complex. This makes the state space explosion problem even more serious than in the non-probabilistic case, and the feasibility of algorithms for quantitative analysis crucially depends on good heuristics to increase efficiency. Hence, model-checking tools usually implement advanced versions of algorithms we present in this chapter, and use intricate data structures to tackle the state space explosion problem, such as multi-terminal binary decision diagrams [54] and sparse matrices. We give a more detailed overview of the implementation approaches in Sect. 28.7.1.

28.1.3 Outline

The remaining sections of this chapter are organized as follows. Section 28.2 presents the definition of Markov decision processes and explains the main concepts that are relevant for PCTL and LTL model checking. The syntax and semantics of PCTL will be provided in Sect. 28.3. Section 28.4 summarizes the main steps of the PCTL model-checking algorithm for MDPs. Section 28.5 introduces the syntax and semantics of LTL and Sect. 28.6 describes the model-checking algorithm. Section 28.7 gives a brief overview of available tools and interesting case studies; it also mentions outstanding challenges of modelling and verification of probabilistic systems. Section 28.8 summarises related models and logics, and Sect. 28.9 concludes the chapter.

28.2 Modelling Probabilistic Concurrent Systems

Markov decision processes [4, 87, 99], which are similar to probabilistic automata [101, 102], are a convenient representation for distributed or concurrent systems in which the system evolution is described by discrete probabilities. Intuitively, a Markov decision process can be understood as a probabilistic variant of a labelled transition system with transitions and states labelled with action labels and atomic

propositions, respectively. For each state s and action α that is enabled in state s, a discrete probability distribution specifies the probabilities for the α-labelled transitions emanating from s. This corresponds to the so-called reactive model in the classification of [55]. In addition, a real-valued cost can be associated with each state s and action α, representing the price one has to pay whenever executing action α in state s. Dually, the cost assigned to (s, α) can also be viewed as a reward that is earned when firing action α in s. To keep the presentation simple, in this chapter we restrict ourselves to cost functions whose range is the non-negative integers. Furthermore, we assume that all transition probabilities in the MDP are rational.

28.2.1 Preliminaries

Let X be a countable set. A *(probability) distribution* on X denotes a function $\mathsf{D} : X \to [0, 1]$ such that

$$\sum_{x \in X} \mathsf{D}(x) = 1.$$

The set $\mathsf{Supp}(\mathsf{D}) \stackrel{\text{def}}{=} \{x \in X : \mathsf{D}(x) \neq 0\}$ is called the support of D. A distribution D is *Dirac* if its support is a singleton. We write $\mathsf{Distr}(X)$ to denote the set of all distributions on X.

As usual, $\mathbb{N}$ denotes the set of natural numbers $0, 1, 2, \ldots$ and $\mathbb{Q}$ the set of rational numbers.

28.2.2 Markov Decision Processes

A *Markov decision process* is a tuple $\mathscr{M} = (S, \mathsf{Act}, \mathsf{P}, s_{\mathsf{init}}, \mathsf{AP}, \mathsf{L}, \mathsf{C})$ where

- S is a countable non-empty set of *states*,
- Act is a finite non-empty set of *actions*,
- $\mathsf{P} : S \times \mathsf{Act} \times S \to [0, 1] \cap \mathbb{Q}$ is the *transition probability function* such that

$$\sum_{s' \in S} \mathsf{P}\big(s, \alpha, s'\big) \in \{0, 1\} \quad \text{for all states } s \in S \text{ and actions } \alpha \in \mathsf{Act},$$

- $s_{\mathsf{init}} \in S$ is the *initial state*,
- AP is a finite set of *atomic propositions*,
- $\mathsf{L} : S \to 2^{\mathsf{AP}}$ is a *labelling function* that labels a state s with those atomic propositions in AP that are supposed to hold in s,
- $\mathsf{C} : S \times \mathsf{Act} \to \mathbb{N}$ is a *cost function*.

$\mathscr{M}$ is called finite if the state space S and the set of actions Act are finite. In this chapter we assume that all MDPs are finite, unless specified otherwise. If $s \in S$ then $\mathsf{Act}(s)$ denotes the set of actions that are *enabled* in state s, i.e.

$$\mathsf{Act}(s) \stackrel{\text{def}}{=} \big\{\alpha \in \mathsf{Act} : \mathsf{P}\big(s, \alpha, s'\big) > 0 \text{ for some } s' \in S\big\}.$$

For technical reasons, we suppose that there are no terminal states, i.e., for each state $s \in S$ the set $\mathsf{Act}(s)$ is non-empty. Furthermore, we require that $\mathsf{C}(s, \alpha) = 0$ if α is an action that is not enabled in s, i.e., if $\alpha \notin \mathsf{Act}(s)$.

The intuitive operational behaviour of an MDP can be described as follows. The MDP starts its computation in the initial state s_{init}. If after n steps the current state is s_n then, first, an enabled action $\alpha_{n+1} \in \mathsf{Act}(s_n)$ is chosen nondeterministically. Firing α_{n+1} in state s_n incurs the cost $\mathsf{C}(s_n, \alpha_{n+1})$. The effect of taking action α_{n+1} in state s_n is given by the distribution $\mathsf{P}(s_n, \alpha_{n+1}, \cdot)$. The next state s_{n+1} belongs to the support of $\mathsf{P}(s_n, \alpha_{n+1}, \cdot)$ and is chosen probabilistically. The resulting infinite sequence of states and actions $\pi = s_0\,\alpha_1\,s_1\,\alpha_2\,s_2\,\alpha_3\,\ldots \in (S \times \mathsf{Act})^\omega$ is called an (infinite) path of $\mathscr{M}$. More generally, any alternating sequence $\pi = s_0\,\alpha_1\,s_1\,\alpha_2\,s_2\,\alpha_3\,\ldots \in (S \times \mathsf{Act})^\omega$, with $\mathsf{P}(s_n, \alpha_{n+1}, s_{n+1}) > 0$ for all $n \geq 0$, is called a *path* of state s_0, and will be written in the form

$$\pi = s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \xrightarrow{\alpha_3} \ldots$$

$\mathsf{Paths}^{\mathscr{M}}(s)$, or for short $\mathsf{Paths}(s)$, denotes the set of all paths of $\mathscr{M}$ starting in state s, and $\mathsf{Paths}^{\mathscr{M}}$, or Paths, denotes the set of all paths. If π is as above then $\pi{\uparrow}^n$ denotes the infinite suffix of π that starts in the $(n{+}1)$-th state s_n, i.e. for the above π we have

$$\pi{\uparrow}^n \stackrel{\text{def}}{=} s_n \xrightarrow{\alpha_{n+1}} s_{n+1} \xrightarrow{\alpha_{n+2}} s_{n+2} \xrightarrow{\alpha_{n+3}} \ldots$$

Similarly, $\pi{\downarrow}_n$ denotes the finite prefix that ends in s_n, i.e.,

$$\pi{\downarrow}_n \stackrel{\text{def}}{=} s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \xrightarrow{\alpha_3} \ldots \xrightarrow{\alpha_n} s_n.$$

We refer to the finite prefixes of (infinite) paths as *finite paths* and denote the set of finite paths starting in state s by $\mathsf{FinPaths}^{\mathscr{M}}(s)$, or for short $\mathsf{FinPaths}(s)$, and we denote the set of all finite paths by $\mathsf{FinPaths}^{\mathscr{M}}$ or $\mathsf{FinPaths}$. The length of a finite path ς is given by the number of transitions taken in ς and denoted by $|\varsigma|$; the length of an infinite path is ω. We use the notation $\mathsf{last}(\varsigma)$ for the last state of a finite path ς. Similarly, $\mathsf{first}(\cdot)$ is used to refer to the first state of a finite or infinite path. The $(n{+}1)$-th state of a path is denoted by $\pi[n]$. Thus, if π is as above then $\pi[0] = \mathsf{first}(\pi) = s_0$, $|\pi{\downarrow}_n| = n$ and $\pi[n] = \mathsf{first}(\pi{\uparrow}^n) = \mathsf{last}(\pi{\downarrow}_n) = s_n$ for all $n \in \mathbb{N}$.

Given a finite path $\varsigma = s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} s_n$, the *total* or *cumulated cost* of ς is defined by

$$\mathsf{cost}(\varsigma) \stackrel{\text{def}}{=} \sum_{i=1}^{n} \mathsf{C}(s_{i-1}, \alpha_i).$$

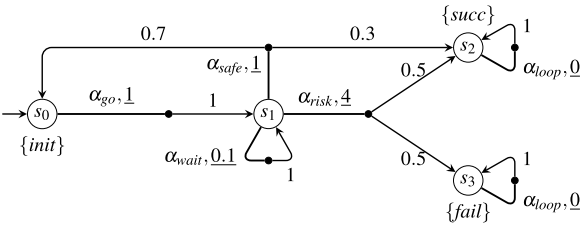


Fig. 1 A running example of a Markov decision process annotated with costs

In addition to the cost function $\mathsf{C}(s, \alpha)$ that assigns values to the pairs consisting of a state and an enabled action, one can also define cost functions just for the states $\mathsf{C}_{\mathsf{st}} : S \to \mathbb{N}$, with the intuitive meaning that each visit to state s incurs the cost $\mathsf{C}_{\mathsf{st}}(s)$. Such cost functions are supported, for example, by the tool PRISM (see Sect. 28.7), but are omitted here since they can be encoded in the variant of MDPs presented in this chapter. If a cost function C_{st} for the states, rather than for pairs of states and actions, is given, then we might switch from C_{st} to $\mathsf{C} : S \times \mathsf{Act} \to \mathbb{N}$ as follows

$$\mathsf{C}(s, \alpha) \stackrel{\text{def}}{=} \begin{cases} \mathsf{C}_{\mathsf{st}}(s) & \text{if } \alpha \in \mathsf{Act}(s) \\ 0 & \text{otherwise} \end{cases}$$

to meet the syntax of the MDP definition. Given an MDP as defined in Sect. 28.2.2 and an additional cost function $\mathsf{C}_{\mathsf{st}} : S \to \mathbb{N}$ that specifies the cost incurred upon visiting state s, the effect of C and C_{st} can be mimicked by using the single cost function $\mathsf{C}' : S \times \mathsf{Act} \to \mathbb{N}$ given by

$$\mathsf{C}'(s, \alpha) \stackrel{\text{def}}{=} \mathsf{C}_{\mathsf{st}}(s) + \mathsf{C}(s, \alpha).$$

Example 1 (Running Example) Consider the MDP $\mathscr{M} = (S, \mathsf{Act}, \mathsf{P}, s_0, \mathsf{AP}, \mathsf{L}, \mathsf{C})$ from Fig. 1. The MDP models a simple system in which, after some initial step, two kinds of decisions can be taken. One results in success with relatively high probability, but can fail completely, and another gives a smaller probability of immediate success, but cannot result in a non-recoverable failure. Formally, $S = \{s_0, s_1, s_2, s_3\}$, $\mathsf{Act} = \{\alpha_{go}, \alpha_{wait}, \alpha_{safe}, \alpha_{risk}, \alpha_{loop}\}$, and P is as given by the numbers on arrows originating from the dots, e.g., $\mathsf{P}(s_1, \alpha_{safe}, s_0) = 0.7$. Atomic propositions are $\{init, succ, fail\}$, where the labels of states are as shown in the picture, e.g., $\mathsf{L}(s_0) = \{init\}$. Costs of the actions are shown in the picture as underlined numbers, e.g., $\mathsf{C}(s_1, \alpha_{wait}) = 0.1$.

Observe that there is a non-trivial choice of an action only in the state s_1, where one can choose between α_{wait}, α_{safe} and α_{risk}. Consider the path

$$\pi = s_0 \xrightarrow{\alpha_{go}} s_1 \xrightarrow{\alpha_{safe}} s_0 \xrightarrow{\alpha_{go}} s_1 \xrightarrow{\alpha_{risk}} s_2 \xrightarrow{\alpha_{loop}} s_2 \xrightarrow{\alpha_{loop}} \cdots .$$

We have $\pi\uparrow^2 = s_0 \xrightarrow{\alpha_{go}} s_1 \xrightarrow{\alpha_{risk}} s_2 \xrightarrow{\alpha_{loop}} s_2 \xrightarrow{\alpha_{loop}} \cdots$ and $\pi\downarrow_2 = s_0 \xrightarrow{\alpha_{go}} s_1 \xrightarrow{\alpha_{safe}} s_0$. For the finite path $\pi\downarrow_2$ we have that the total or cumulated cost $\mathsf{cost}(\pi\downarrow_2) = \mathsf{C}(s_0, \alpha_{go}) + \mathsf{C}(s_1, \alpha_{safe}) = 2$. □

28.2.3 Markov Chains

Markov chains can be viewed as special instances of Markov decision processes, where in each state exactly one action is enabled. Thus, there are no nondeterministic choices in a Markov chain and the operational behaviour is purely probabilistic. Since, in the above definition of an MDP, the actions are used just to name the nondeterministic alternatives and group together probabilistic transitions that belong to the same alternative, the concept of actions is irrelevant for Markov chains. Thus, the transition probabilities of a Markov chain $\mathscr{C}$ can be specified by a function $\mathsf{P}^{\mathscr{C}} : S \times S \to [0, 1]$. Paths are then just sequences $s_0\, s_1\, s_2 \ldots$ of states such that

$$\mathsf{P}^{\mathscr{C}}(s_i, s_{i+1}) > 0 \quad \text{for all } i \geq 0.$$

Using standard concepts of measure and probability theory, any Markov chain naturally induces a *probability space*, i.e., a triple consisting of the set of *outcomes* Ω, the set of *events* $\mathscr{F} \subseteq 2^{\Omega}$ which contains $\emptyset$ and is closed under complements and countable unions, and a *probability measure* $\mathsf{Pr} : \mathscr{F} \to [0, 1]$ which is countably additive and satisfies $\mathsf{Pr}(\Omega) = 1$. More concretely, in the induced probability space the outcomes are the (infinite) paths and the events can be understood as linear-time properties, i.e., conditions that an infinite path might satisfy or not (indeed, all LTL formulas, PCTL path formulas, and even all ω-regular languages over sets of atomic propositions specify measurable sets of paths [40, 105]). For details we refer to textbooks on Markov chains and probability theory, see, for example, [50, 75, 77], and just sketch the main ideas. The underlying σ-algebra is the smallest σ-algebra that contains the *cylinder sets*, namely, the sets containing all paths that have a common prefix, i.e., the sets

$$\mathsf{Cyl}(\varsigma) \stackrel{\text{def}}{=} \left\{\pi \in \mathsf{Paths}^{\mathscr{C}} : \varsigma \text{ is a prefix of } \pi\right\}$$

for all finite paths ς in $\mathscr{C}$. Using Carathéodory's measure extension theorem [7], the probability measure $\mathsf{Pr}^{\mathscr{C}}$ is the unique probability measure on the σ-algebra such that for each finite path $\varsigma = s_0\, s_1\, s_2 \ldots s_n$ starting in $\mathscr{C}$'s initial state $s_0 = s_{\mathsf{init}}$ we have:

$$\mathsf{Pr}^{\mathscr{C}}\big(\mathsf{Cyl}(\varsigma)\big) = \mathsf{P}^{\mathscr{C}}(s_0, s_1) \cdot \mathsf{P}^{\mathscr{C}}(s_1, s_2) \cdot \ldots \cdot \mathsf{P}^{\mathscr{C}}(s_{n-1}, s_n).$$

If ς is a finite path that does not start in the initial state then $\mathsf{Pr}^{\mathscr{C}}(\mathsf{Cyl}(\varsigma)) = 0$.

28.2.4 Schedulers

Reasoning about probabilities in an MDP relies on a *decision-making* approach that resolves the nondeterministic choices—answering the question which action will be performed in the current state—and turns an MDP into an infinite tree-like Markov

chain. We give here just a brief summary of the main concepts. Details can be found in any textbook on Markov decision processes, e.g., [99].

The decision-making approach can be formalized with the help of the mathematical notion of a *scheduler*, often called *policy* or *adversary*. Intuitively, a scheduler takes as input the "history" of a computation—namely, a finite path ς—and chooses the next action according to some distribution. Formally, a *history-dependent randomized* scheduler, for short called a scheduler, is a function

$$\mathscr{U} : \mathsf{FinPaths}^{\mathscr{M}} \to \mathsf{Distr}(\mathsf{Act})$$

such that $\mathsf{Supp}(\mathscr{U}(\varsigma)) \subseteq \mathsf{Act}(\mathsf{last}(\varsigma))$ for all finite paths ς. A (finite or infinite) path $\pi = s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \ldots$ is said to be a $\mathscr{U}$-path, if

$$\mathscr{U}(s_0 \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_i} s_i)(\alpha_{i+1}) > 0 \quad \text{for all } 0 \leq i < |\pi|.$$

A scheduler $\mathscr{U}$ is called *deterministic* if $\mathscr{U}(\varsigma)$ is a Dirac distribution for all finite paths ς, i.e., for each finite path ς there is some action α with $\mathscr{U}(\varsigma)(\alpha) = 1$, and $\mathscr{U}(\varsigma)(\beta) = 0$ for all actions $\beta \in \mathsf{Act} \setminus \{\alpha\}$. Scheduler $\mathscr{U}$ is called *memoryless* if

$$\mathscr{U}(\varsigma) = \mathscr{U}(\varsigma') \text{ for all finite paths } \varsigma, \varsigma' \text{ such that } \mathsf{last}(\varsigma) = \mathsf{last}(\varsigma').$$

Deterministic schedulers are given as functions $\mathscr{U} : \mathsf{FinPaths}^{\mathscr{M}} \to \mathsf{Act}$. Memoryless randomized schedulers can be viewed as functions $\mathscr{U} : S \to \mathsf{Distr}(\mathsf{Act})$. Memoryless deterministic schedulers, also called *simple schedulers*, are specified as functions $\mathscr{U} : S \to \mathsf{Act}$. We write Sched to denote the set of all schedulers.

28.2.5 Probability Measures in MDPs

Given an MDP $\mathscr{M}$ and a scheduler $\mathscr{U}$, the behaviour of $\mathscr{M}$ under $\mathscr{U}$ can be formalized by an infinite-state tree-like Markov chain $\mathscr{C} = \mathscr{M}|_{\mathscr{U}}$. The states of that Markov chain represent the finite $\mathscr{U}$-paths. The successor states of

$$\varsigma = s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} s_n$$

have the form $\varsigma' = \varsigma \xrightarrow{\beta} s$ and the transition probability for moving from ς to ς' is given by

$$\mathscr{U}(\varsigma)(\beta) \cdot \mathsf{P}(s_n, \beta, s).$$

We write $\mathsf{Pr}^{\mathscr{M},\mathscr{U}}$, or for short $\mathsf{Pr}^{\mathscr{U}}$, to denote the standard probability measure $\mathsf{Pr}^{\mathscr{C}}$ on that Markov chain. Thus, the probability measure $\mathsf{Pr}^{\mathscr{U}}$ for a given scheduler $\mathscr{U}$ is the unique probability measure on the σ-algebra generated by the finite $\mathscr{U}$-paths such that

$$\mathsf{Pr}^{\mathscr{U}}\big(\mathsf{Cyl}(\varsigma)\big) = \prod_{i=1}^{n} \mathscr{U}(\varsigma \downarrow_{i-1})(\alpha_i) \cdot \mathsf{P}(s_{i-1}, \alpha_i, s_i)$$

if $\varsigma = s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} s_n$ is a $\mathscr{U}$-path starting in $s_0 = s_{\mathsf{init}}$.

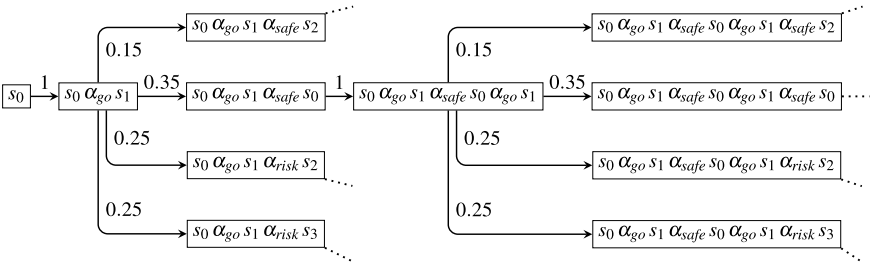


Fig. 2 A Markov chain for the running example and the scheduler from Example 2

Given a state s of $\mathscr{M}$, we denote by $\Pr_s^{\mathscr{U}}$ the probability measure that is obtained by $\mathscr{U}$ viewed as a scheduler for the MDP $\mathscr{M}_s$ that agrees with $\mathscr{M}$, except that s is the unique initial state of $\mathscr{M}_s$. That is, if $\mathscr{M} = (S, \mathsf{Act}, \mathsf{P}, s_{\mathsf{init}}, \mathsf{AP}, \mathsf{L}, \mathsf{C})$ then $\mathscr{M}_s = (S, \mathsf{Act}, \mathsf{P}, s, \mathsf{AP}, \mathsf{L}, \mathsf{C})$. Note that if $\mathscr{U}$ is a deterministic scheduler then

$$\Pr_s^{\mathscr{U}}\big(\mathsf{Cyl}(\varsigma)\big) = \prod_{i=1}^{n} \mathsf{P}(s_{i-1}, \alpha_i, s_i)$$

if $\varsigma = s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} s_n$ is a $\mathscr{U}$-path with $\mathsf{first}(\varsigma) = s_0 = s$. Given an MDP $\mathscr{M}$, a scheduler $\mathscr{U}$ and a measurable path property E, then

$$\Pr_s^{\mathscr{U}}(E) \stackrel{\text{def}}{=} \Pr_s^{\mathscr{U}}\big\{\pi \in \mathsf{Paths}^{\mathscr{M}} \;\big|\; \pi \text{ satisfies } E\big\}$$

denotes the probability that the path property E holds in $\mathscr{M}$ when starting in s and using scheduler $\mathscr{U}$ to resolve the nondeterministic choices.

Example 2 Consider again the MDP $\mathscr{M}$ from Fig. 1, together with the scheduler $\mathscr{U}$ that for every path ending in s_1 picks the action α_{safe} or α_{risk}, both with probability 0.5. This scheduler is memoryless, but not deterministic, and gives rise to the Markov chain $\mathscr{M}|_{\mathscr{U}}$ whose initial fragment is drawn in Fig. 2. For the finite path $\pi = s_0 \xrightarrow{\alpha_{go}} s_1 \xrightarrow{\alpha_{safe}} s_0$ we have

$$\begin{aligned}\Pr^{\mathscr{U}}\big(\mathsf{Cyl}(\pi)\big) &= \mathscr{U}(s_0)(\alpha_{go}) \cdot \mathsf{P}(s_0, \alpha_{go}, s_1) \cdot \mathscr{U}(s_0 \xrightarrow{\alpha_{go}} s_1)(\alpha_{safe}) \cdot \mathsf{P}(s_1, \alpha_{safe}, s_0) \\ &= 1 \cdot 1 \cdot 0.5 \cdot 0.7 = 0.35,\end{aligned}$$

and for the set of paths R which never reach s_2 or s_3 we have $\Pr^{\mathscr{U}}(R) = 0$. □

28.2.6 Maximal and Minimal Probabilities for Path Events

A typical task for the quantitative analysis of an MDP is to compute minimal or maximal probabilities for some given property E when ranging over all schedulers.

If s is a state in $\mathscr{M}$ then we define

$$\mathsf{Pr}_s^{\max}(E) \stackrel{\text{def}}{=} \sup_{\mathscr{U} \in \mathsf{Sched}} \mathsf{Pr}_s^{\mathscr{U}}(E) \quad \text{and} \quad \mathsf{Pr}_s^{\min}(E) \stackrel{\text{def}}{=} \inf_{\mathscr{U} \in \mathsf{Sched}} \mathsf{Pr}_s^{\mathscr{U}}(E).$$

This corresponds to the worst- or best-case analysis of an MDP. If, for example, E stands for the undesired behaviours then E is guaranteed not to hold with probability at least $1 - \mathsf{Pr}_s^{\max}(E)$ under all schedulers, that is, even for the worst-case resolution of the nondeterministic choices. For instance, many relevant properties fall under the class of *reachability probabilities* where one has to establish a lower bound for the minimal probability to reach a certain set F of "good" target states, possibly with some side-constraints on the cumulated cost until an F-state has been reached.

28.2.7 Maximal and Minimal Expected Cost

Another typical task for analysing an MDP against cost-based properties is to compute the minimal or maximal *expected cumulated cost* with respect to certain objectives. For reachability objectives, we consider a set F of target states. Given a path $\pi = s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \xrightarrow{\alpha_3} \ldots$, we write $\pi \models \Diamond F$ if and only if π eventually visits F, i.e., there is an i such that $s_i \in F$. The cumulated cost of π to reach F is defined as follows. If $\pi \models \Diamond F$ then

$$\mathsf{cost}[\Diamond F](\pi) = \mathsf{cost}(\pi{\downarrow_n}) = \sum_{i=1}^{n} \mathsf{C}(s_{i-1}, \alpha_i)$$

where $s_n \in F$ and $\{s_i : 0 \leq i < n\} \cap F = \varnothing$. If π never visits a state in F then $\mathsf{cost}[\Diamond F](\pi)$ is defined as ∞, irrespective of whether only finitely many actions in π have nonzero cost (in which case the total cost of π would be finite). Given a scheduler $\mathscr{U}$ for $\mathscr{M}$ and a state s in $\mathscr{M}$, the expected cumulated cost for reaching F from s, denoted $\mathsf{Ex}_s^{\mathscr{U}}(\mathsf{cost}[\Diamond F])$, is the expected value of the random variable $\pi \mapsto \mathsf{cost}[\Diamond F](\pi)$ in the stochastic process (i.e., the Markov chain) induced by $\mathscr{U}$.

- If $\mathsf{Pr}_s^{\mathscr{U}}(\{\pi \in \mathsf{Paths} \mid \pi \models \Diamond F\}) = 1$ then

$$\mathsf{Ex}_s^{\mathscr{U}}\big(\mathsf{cost}[\Diamond F]\big) = \sum_{\varsigma} \mathsf{Pr}_s^{\mathscr{U}}\big(\mathsf{Cyl}(\varsigma)\big) \cdot \mathsf{cost}(\varsigma)$$

  where the sum is taken over all finite $\mathscr{U}$-paths ς with $\mathsf{first}(\varsigma) = s$ and $\mathsf{last}(\varsigma) \in F$, while all other states of ς are in $S \setminus F$.
- If $\mathsf{Pr}_s^{\mathscr{U}}(\{\pi \in \mathsf{Paths} \mid \pi \models \Diamond F\}) < 1$ then with positive probability $\mathscr{U}$ schedules paths that never visit F. Since the total cost of such paths is infinite, we have $\mathsf{Ex}_s^{\mathscr{U}}(\mathsf{cost}[\Diamond F]) = \infty$.

The extremal expected cumulated cost for reaching F is then obtained by

$$\mathsf{Ex}_s^{\max}\big(\mathsf{cost}[\Diamond F]\big) \stackrel{\text{def}}{=} \sup_{\mathscr{U}\in\mathsf{Sched}} \mathsf{Ex}_s^{\mathscr{U}}\big(\mathsf{cost}[\Diamond F]\big)$$

$$\mathsf{Ex}_s^{\min}\big(\mathsf{cost}[\Diamond F]\big) \stackrel{\text{def}}{=} \inf_{\mathscr{U}\in\mathsf{Sched}} \mathsf{Ex}_s^{\mathscr{U}}\big(\mathsf{cost}[\Diamond F]\big).$$

Note that $\mathsf{Ex}_s^{\max}(\mathsf{cost}[\Diamond F]) = \infty$ if $\mathsf{Pr}_s^{\min}(\{\pi \in \mathsf{Paths} \mid \pi \models \Diamond F\}) < 1$; the other direction also holds, i.e., $\mathsf{Pr}_s^{\min}(\{\pi \in \mathsf{Paths} \mid \pi \models \Diamond F\}) = 1$ implies that $\mathsf{Ex}_s^{\max}(\mathsf{cost}[\Diamond F])$ is finite, although the proof is not as obvious.

Similarly, minimal and maximal expected cost for other objectives can be defined. If an MDP is used as a discrete-time model then one might be interested in the average cost within certain time intervals. This, for instance, permits us to establish lower or upper bounds on the expected power consumption over one time unit. For the *cost cumulated up to time point k* we use the random variable $\pi \mapsto \mathsf{cost}[{\leq}k](\pi)$ that assigns to each path the total cost for the first k steps, i.e., if $\pi = s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \ldots$ then

$$\mathsf{cost}[{\leq}k](\pi) \stackrel{\text{def}}{=} \sum_{i=1}^{k} \mathsf{C}(s_{i-1}, \alpha_i).$$

Let $\mathsf{Ex}_s^{\mathscr{U}}(\mathsf{cost}[{\leq}k])$ denote the expected value of the random variable $\mathsf{cost}[{\leq}k]$ under scheduler $\mathscr{U}$ in the MDP $\mathscr{M}_s$, i.e.,

$$\mathsf{Ex}_s^{\mathscr{U}}\big(\mathsf{cost}[{\leq}k]\big) = \sum_{\varsigma} \mathsf{Pr}_s^{\mathscr{U}}\big(\mathsf{Cyl}(\varsigma)\big) \cdot \mathsf{cost}(\varsigma)$$

where the sum is taken over all finite $\mathscr{U}$-paths ς of length k starting in state s. The supremum and infimum over all schedulers yields the extremal cumulated costs within the first k steps

$$\mathsf{Ex}_s^{\max}\big(\mathsf{cost}[{\leq}k]\big) \stackrel{\text{def}}{=} \sup_{\mathscr{U}\in\mathsf{Sched}} \mathsf{Ex}_s^{\mathscr{U}}\big(\mathsf{cost}[{\leq}k]\big)$$

$$\mathsf{Ex}_s^{\min}\big(\mathsf{cost}[{\leq}k]\big) \stackrel{\text{def}}{=} \inf_{\mathscr{U}\in\mathsf{Sched}} \mathsf{Ex}_s^{\mathscr{U}}\big(\mathsf{cost}[{\leq}k]\big).$$

When we specify costs for the states by the function $\mathsf{C}_{\mathsf{st}} : S \to \mathbb{N}$, then it is also possible to reason about *instantaneous costs* in the k-th step. This can be defined with the random variable $\pi \mapsto \mathsf{cost}[{=}k](\pi)$ that assigns to each path π the cost associated with the k-th action of π. If $\mathsf{Ex}_s^{\mathscr{U}}(\mathsf{cost}[{=}k])$ denotes the expected value of random variable $\mathsf{cost}[{=}k]$ under scheduler $\mathscr{U}$ then

$$\mathsf{Ex}_s^{\max}\big(\mathsf{cost}[{=}k]\big) = \sup_{\mathscr{U}\in\mathsf{Sched}} \mathsf{Ex}_s^{\mathscr{U}}\big(\mathsf{cost}[{=}k]\big)$$

$$\mathsf{Ex}_s^{\min}\big(\mathsf{cost}[{=}k]\big) = \inf_{\mathscr{U}\in\mathsf{Sched}} \mathsf{Ex}_s^{\mathscr{U}}\big(\mathsf{cost}[{=}k]\big)$$

stand for the extremal average instantaneous costs incurred at the k-th step. These values can be of interest, for example, when reasoning about the minimal or maximal expected queue size at some time point k. For this purpose, we work with the cost function $\mathsf{C}(t, \alpha) = \mathsf{C}_{\mathsf{st}}(t)$ for all actions α that are enabled in state t, where $\mathsf{C}_{\mathsf{st}}(t)$ denotes the current queue size in state t.

Example 3 Let us return to our running example from Fig. 1, and for clarity of notation write just s instead of the singleton set $\{s\}$. We have that the maximal probability of reaching s_3, i.e., $\mathsf{Pr}^{\max}_{s_0}(\{\pi \in \mathsf{Paths} \mid \pi \models \Diamond s_3\})$, is equal to 0.5. A (deterministic) scheduler that always chooses α_{risk} in paths ending with s_1 witnesses that $\mathsf{Pr}^{\max}_{s_0}(\{\pi \in \mathsf{Paths} \mid \pi \models \Diamond s_3\}) \geq 0.5$; to see that this probability cannot be higher, observe that upon taking α_{risk} half of the paths transition to s_2, and both s_2 and s_3 have self-loops. On the other hand, $\mathsf{Pr}^{\min}_{s_0}(\{\pi \in \mathsf{Paths} \mid \pi \models \Diamond s_3\}) = 0$, as witnessed by the scheduler that never chooses α_{risk} with nonzero probability.

For maximal expected cost, let us consider a single target state s_2. We have $\mathsf{Ex}^{\max}_{s_0}(\mathsf{cost}[\Diamond s_2]) = \infty$, because there exists a scheduler that with nonzero probability does not reach s_2. For minimal expected cost $\mathsf{Ex}^{\min}_{s_0}(\mathsf{cost}[\Diamond s_2])$, we obtain the value equal to $\frac{20}{3}$, as witnessed by the scheduler that always chooses α_{safe}; to see that no scheduler can yield a better value is a simple exercise.

As an example of instantaneous cost, let us analyse the value $\mathsf{Ex}^{\max}_{s_0}(\mathsf{cost}[=3])$. It is equal to 4, which can be seen by considering a scheduler that picks α_{wait} in $s_0 \xrightarrow{\alpha_{go}} s_1$, and α_{risk} in $s_0 \xrightarrow{\alpha_{go}} s_1 \xrightarrow{\alpha_{wait}} s_1$. This is also the maximal value, because there is in fact no higher cost in the MDP.

28.3 Probabilistic Computation Tree Logic

In this section we present the syntax and semantics of Probabilistic Computation Tree Logic (PCTL), which is a probabilistic counterpart of the well-known logic CTL, introduced in Chap. 2. Formulas of this logic aim to express quantitative probabilistic properties such as "with probability at least 0.99, if we reach a bad state, we can recover with nonzero probability". PCTL is a widely used specification language in many contexts, including verification of purely probabilistic systems or systems with probability as well as nondeterminism, and for both finite- and infinite-state probabilistic systems [15, 22, 81]. Our presentation will focus on the logic PCTL interpreted over finite-state Markov decision processes.

28.3.1 Syntax of PCTL

As in CTL, the syntax of PCTL has two levels: one for the state formulas (denoted by uppercase Greek letters Φ, Ψ) and one for the path formulas (denoted by lower-

case Greek letters φ, ψ). The abstract syntax of state and path formulas is as follows

$$\Phi ::= \mathsf{tt} \mid a \mid \Phi_1 \wedge \Phi_2 \mid \neg\Phi \mid \mathbb{P}_{\sim p}(\varphi) \mid \mathbb{E}_{\sim c}(\Diamond\Phi) \mid \mathbb{E}_{\sim c}(\leq k) \mid \mathbb{E}_{\sim c}(=k)$$

$$\varphi ::= \bigcirc\Phi \mid \Phi_1 \mathsf{U} \Phi_2 \mid \Phi_1 \mathsf{U}^{\sim c} \Phi_2$$

where tt stands for the constant truth value "true" and a is a state predicate, i.e., an atomic proposition in AP. The other symbols are explained below.

The operators $\mathbb{P}_{\sim p}(\cdot)$ and $\mathbb{E}_{\sim c}(\cdot)$ are called the *probability* and *expectation* operators. The subscripts $\sim p$ and $\sim c$ specify strict or non-strict lower or upper bounds for probabilities or costs, respectively. Formally, $\sim$ is a comparison operator $\leq$, $<$, $\geq$ or $>$, $p \in [0, 1] \cap \mathbb{Q}$ a rational threshold for probabilities, and $c \in \mathbb{N}$ a non-negative integer that serves as a lower or upper bound for cumulated or instantaneous cost.

The PCTL state formula $\mathbb{P}_{\sim p}(\varphi)$ asserts that, under all schedulers, the probability for the event expressed by the path formula φ meets the bound specified by $\sim p$. Thus, the probability operator imposes a condition on the probability measures $\Pr_s^{\mathscr{U}}$ for all schedulers $\mathscr{U}$. The probability bounds "$\sim p$" can be understood as quantitative counterparts to the CTL path quantifiers $\exists$ and $\forall$. Intuitively, the lower probability bounds $\geq p$ (with $p > 0$) or $> p$ (with $p \geq 0$) can be understood as the quantitative counterpart to existential path quantification. (See also Remark 1.)

As in CTL, path formulas are built from one of the temporal modalities $\bigcirc$ (next) or U (until), where the arguments of the modalities are state formulas. No Boolean connectors or nesting of temporal modalities are allowed in the syntax of path formulas. In addition to the standard until-operator, the above syntax for path formulas includes a cost-bounded version of until.[1] The intuitive meaning of the path formula $\Phi_1 \mathsf{U}^{\sim c} \Phi_2$ is that a Φ_2-state (i.e., some state where Φ_2 holds) will be reached from the current state along a finite path ς that yields a witness of minimal length for the path formula $\Phi_1 \mathsf{U} \Phi_2$ (i.e., ς ends in a Φ_2-state and all other states satisfy the formula $\Phi_1 \wedge \neg\Phi_2$) and where the total cost of ς meets the constraint $\sim c$.

The expectation operator $\mathbb{E}_{\sim c}(\cdot)$ enables the specification of lower or upper bounds for the expected cumulated or instantaneous cost. The state formula $\mathbb{E}_{\sim c}(\Diamond\Phi)$ holds if the expected cumulated cost until a Φ-state is reached meets the requirement given by "$\sim c$" under all schedulers. Similarly, the state formulas $\mathbb{E}_{\sim c}(\leq k)$ and $\mathbb{E}_{\sim c}(=k)$ assert that the cost accumulated in the first k steps and the instantaneous cost at the k-th step, respectively, belong to the interval specified by "$\sim c$".

28.3.2 Semantics of PCTL

Given an MDP, the satisfaction relation $\models$ for state and path formulas is formally defined below, in accordance with the above intuitive semantics. Let $\mathscr{M}$ be an MDP

[1] We did not introduce the step-bounded version of the until operator. This, however, can be derived using the cost-bounded until operator and changing the MDP to the one with unit cost, i.e., $\mathsf{C}(s, \alpha) = 1$ for all states s and actions $\alpha \in \mathsf{Act}(s)$.

as in Sect. 28.2.2 and s a state in $\mathscr{M}$.

$$
\begin{array}{lll}
s \models \mathsf{tt} & & \\
s \models a & \text{iff} & a \in \mathsf{L}(s) \\
s \models \Phi_1 \wedge \Phi_2 & \text{iff} & s \models \Phi_1 \text{ and } s \models \Phi_2 \\
s \models \neg\Phi & \text{iff} & s \not\models \Phi \\
s \models \mathbb{P}_{\sim p}(\varphi) & \text{iff} & \mathsf{Pr}_s^{\mathscr{U}}(\varphi) \sim p \text{ for all schedulers } \mathscr{U} \\
 & & \text{where } \mathsf{Pr}_s^{\mathscr{U}}(\varphi) \stackrel{\text{def}}{=} \mathsf{Pr}_s^{\mathscr{U}}\{\pi \in \mathsf{Paths} \mid \pi \models \varphi\} \\
s \models \mathbb{E}_{\sim c}(\Diamond\Phi) & \text{iff} & \mathsf{Ex}_s^{\mathscr{U}}(\mathsf{cost}[\Diamond\mathsf{Sat}(\Phi)]) \sim c \text{ for all schedulers } \mathscr{U} \\
 & & \text{where } \mathsf{Sat}(\Phi) \stackrel{\text{def}}{=} \{s \in S \mid s \models \Phi\} \\
s \models \mathbb{E}_{\sim c}(\leq k) & \text{iff} & \mathsf{Ex}_s^{\mathscr{U}}(\mathsf{cost}[\leq k]) \sim c \text{ for all schedulers } \mathscr{U} \\
s \models \mathbb{E}_{\sim c}(= k) & \text{iff} & \mathsf{Ex}_s^{\mathscr{U}}(\mathsf{cost}[= k]) \sim c \text{ for all schedulers } \mathscr{U}
\end{array}
$$

MDP $\mathscr{M}$ is said to satisfy a PCTL state formula Φ, denoted $\mathscr{M} \models \Phi$, if $s_{\text{init}} \models \Phi$. The semantics of the next- and until-operators is exactly as in CTL. If $\pi = s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \xrightarrow{\alpha_3} \ldots$ is an infinite path in $\mathscr{M}$ then

$$
\begin{array}{lll}
\pi \models \bigcirc\Phi & \text{iff} & s_1 \models \Phi \\
\pi \models \Phi_1 \mathsf{U} \Phi_2 & \text{iff} & \text{there exists } k \in \mathbb{N} \text{ with } s_k \models \Phi_2 \text{ and } s_i \models \Phi_1 \text{ for all } 0 \leq i < k.
\end{array}
$$

The semantics of the cost-bounded until-operator is as for the standard until-operator, except that we require that the shortest prefix of π that ends in a Φ_2-state meets the cost-bound. Formally,

$$
\begin{array}{lll}
\pi \models \Phi_1 \mathsf{U}^{\sim c} \Phi_2 & \text{iff} & \text{there exists } k \in \mathbb{N} \text{ such that} \\
 & & (1)\ \ s_k \models \Phi_2 \\
 & & (2)\ \ s_i \models \Phi_1 \wedge \neg\Phi_2 \text{ for all } 0 \leq i < k \\
 & & (3)\ \ \mathsf{cost}(\pi{\downarrow_k}) \sim c.
\end{array}
$$

We now justify the above definitions. First, using [40, 105] we get that the set consisting of all paths where a PCTL path formula holds is indeed measurable. Second, we observe that

$$
\begin{array}{lll}
s \models \mathbb{P}_{\leq p}(\varphi) & \text{iff} & \mathsf{Pr}_s^{\max}\{\pi \in \mathsf{Paths} \mid \pi \models \varphi\} \leq p \\
s \models \mathbb{P}_{< p}(\varphi) & \text{iff} & \mathsf{Pr}_s^{\max}\{\pi \in \mathsf{Paths} \mid \pi \models \varphi\} < p.
\end{array}
$$

The first statement is obvious. The second statement follows from the fact that, for the events that can be specified by some PCTL path formula φ, there exists a scheduler that maximizes the probability for φ, and so the supremum defining $\mathsf{Pr}_s^{\max}$ can in fact be replaced with the maximum (see, e.g., [99]). For the next- and unbounded until-operators such a scheduler can in fact be assumed to be simple.

An analogous statement holds for strict or non-strict lower probability bounds and $\mathsf{Pr}_s^{\min}$ rather than $\mathsf{Pr}_s^{\max}$. Similarly, we have

$$s \models \mathbb{E}_{\leq c}(C) \quad \text{iff} \quad \mathsf{Ex}_s^{\max}(C) \leq c$$
$$s \models \mathbb{E}_{<c}(C) \quad \text{iff} \quad \mathsf{Ex}_s^{\max}(C) < c$$

and the analogous statement for lower cost bounds, where C stands for one of the three options $\Diamond\Phi$, $\leq k$, or $=k$. Here, again, minimal or maximal expected cost for the random variable associated with C can be achieved by some scheduler, and in the case of $\Diamond\Phi$ simple schedulers suffice.

Although the above semantics of the probabilistic and expectation operators relies on universal quantification over all schedulers, the existence of at least one scheduler satisfying a certain condition can be expressed using negation in front of the operator. For instance, $\neg\mathbb{P}_{\leq p}(\varphi)$ asserts the existence of a scheduler $\mathscr{U}$ where φ holds with probability $> p$.

Since probabilities are always values in the interval $[0, 1]$, there are some trivial combinations of $\sim$ and p. For instance, $\mathbb{P}_{\geq 0}(\varphi)$ and $\mathbb{P}_{\leq 1}(\varphi)$ are tautologies, while $\mathbb{P}_{<0}(\varphi)$ and $\mathbb{P}_{>1}(\varphi)$ are not satisfiable. In what follows, we write $\mathbb{P}_{=1}(\varphi)$ for $\mathbb{P}_{\geq 1}(\varphi)$ and $\mathbb{P}_{=0}(\varphi)$ for $\mathbb{P}_{\leq 0}(\varphi)$. Similarly, as the cost function assigns non-negative cost to all transitions, the total cost can never be negative. Hence, formulas of the form $\mathbb{E}_{<0}(\cdot)$ are not satisfiable.

28.3.3 Derived Operators

Other Boolean operators can be derived from negation and conjunction as usual, e.g.,

$$\mathsf{ff} \stackrel{\text{def}}{=} \neg\mathsf{tt} \quad \text{and} \quad \Phi_1 \vee \Phi_2 \stackrel{\text{def}}{=} \neg(\neg\Phi_1 \wedge \neg\Phi_2).$$

The eventually operator $\Diamond$, a modality for path formulas, can be obtained as in CTL or LTL by

$$\Diamond\Phi \stackrel{\text{def}}{=} \mathsf{tt}\,\mathsf{U}\,\Phi,$$

and an analogous definition can be derived for the cost-bounded variant

$$\Diamond^{\sim c}\Phi \stackrel{\text{def}}{=} \mathsf{tt}\,\mathsf{U}^{\sim c}\,\Phi.$$

The always operator $\Box$ and its cost-bounded variant $\Box^{\sim c}$ can be derived using the duality of lower and upper probability bounds. For instance, $\mathbb{P}_{\leq p}(\Box\Phi)$ can be defined as $\mathbb{P}_{\geq 1-p}(\Diamond\neg\Phi)$, and $\mathbb{P}_{>p}(\Box^{\sim c}\Phi)$ as $\mathbb{P}_{<1-p}(\Diamond^{\sim c}\neg\Phi)$.

Example 4 (PCTL Formulas for the Running Example) First, we give examples of properties expressible in PCTL. The property "with probability at least 0.99, whenever we reach a bad state, we can recover with nonzero probability" from the beginning of this section can be stated as the formula $\mathbb{P}_{\geq 0.99}(\Box(bad \rightarrow \mathbb{P}_{>0}\Diamond\neg bad))$. Another property is "the expected energy consumption in the first 100 steps is at most 20 units", which is expressed by $\mathbb{E}_{\leq 20}(\leq 100)$, assuming that the relevant cost function quantifies the energy consumed at every step. Further, the formula $\mathbb{P}_{\leq 0.1}(\neg initialised \,\mathsf{U}\, request)$ states that the probability of a request being made before the system initialisation phase completes is at most 0.1.

Now, let us return to the MDP from Example 1 to analyse some PCTL formulas more thoroughly. Consider the formula $\Phi \equiv \mathbb{P}_{\leq 0.6}(\neg succ \mathsf{U}^{\leq 5} fail)$. First, observe that the formula $\neg succ$ holds in the states s_0, s_1 and s_3, whereas the formula *fail* holds only in the state s_3. Paths that satisfy $\neg succ \mathsf{U}^{\leq 5} fail$ are exactly the paths that reach s_3 and whose cost is at most 5. It is easy to see that the probability of these paths is maximal under any scheduler that always chooses α_{risk} deterministically, in which case these paths have probability 0.5. Thus, for any $\mathscr{U}$, we have $\mathsf{Pr}^{\mathscr{U}}_{s_0}(\neg succ \mathsf{U}^{\leq 5} fail) \leq 0.6$ and the formula Φ is satisfied.

On the other hand, the formula $\mathbb{E}_{\leq 5}(\leq 4)$ is not satisfied. Consider, for example, the scheduler that chooses α_{safe} in the path $s_0 \xrightarrow{\alpha_{go}} s_1$ and α_{risk} in the path $s_0 \xrightarrow{\alpha_{go}} s_1 \xrightarrow{\alpha_{safe}} s_0 \xrightarrow{\alpha_{go}} s_1$. Under this scheduler, the expected cost cumulated in 4 steps is 5.5, whereas the required upper bound is 5.

Remark 1 (Qualitative Properties) The conditions imposed by PCTL formulas of the form $\mathbb{P}_{>0}(\varphi)$ or $\mathbb{P}_{=1}(\varphi)$ are often called *qualitative properties*. Their meaning is quite close to CTL formulas $\exists\varphi$ and $\forall\varphi$ which are defined to be true if and only if for every scheduler $\mathscr{U}$ there is a $\mathscr{U}$-path satisfying φ (resp. all $\mathscr{U}$-paths satisfy φ in the case of $\forall\varphi$).

Indeed, if φ is a CTL path formula of the form $\bigcirc a$, $a \mathsf{U} b$ or $a \mathsf{U}^{\sim c} b$ where a, b are atomic propositions, then the PCTL formula $\mathbb{P}_{>0}(\varphi)$ is equivalent to the CTL formula $\exists\varphi$ (interpreted as described above). This is a consequence of the observation that the set of paths where φ holds can be written as a disjoint union of cylinder sets, and hence the requirement to have at least one path π with $\pi \models \varphi$ is equivalent to the requirement that the probability measure of the paths that satisfy φ is positive. Similarly, the PCTL formula $\mathbb{P}_{=1}(\Box a)$ and the CTL formula $\forall\Box a$ are equivalent: if there is a path $\pi = s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \xrightarrow{\alpha_3} \ldots$ where some s_i does not satisfy a, then no path starting with $s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \xrightarrow{\alpha_3} \ldots s_i$ satisfies $\Box a$, and so the probability of paths satisfying $\Box a$ is strictly lower than 1. The same equality holds for $\mathbb{P}_{=1}(\bigcirc a)$ and $\forall \bigcirc a$.

However, there is a mild difference between the meaning of the PCTL formula $\mathbb{P}_{=1}(\Diamond a)$ and the CTL formula $\forall\Diamond a$, because the quantification over *"all paths"* is more restrictive than that over *"almost all paths"* in the case of reachability. Observe that state s satisfies the CTL formula $\forall\Diamond a$ if and only if all paths starting from s will eventually enter an a-state (i.e., a state s' with $s' \models a$). Satisfaction of the PCTL formula $\mathbb{P}_{=1}(\Diamond a)$ in state s means that almost all paths will eventually

visit an a-state, in the sense that the probability measure of the paths π starting in s and satisfying φ equals 1; this includes paths that never enter an a-state, as long as their total probability measure is zero. □

28.4 Model-Checking Algorithms for MDPs and PCTL

We now present an algorithm that, given a PCTL state formula and a Markov decision process, decides whether the formula holds in the MDP or not. The algorithm, similarly to the algorithm for CTL model checking from Chap. 2, consists of separate subprocedures for each (temporal or Boolean) connective. Instead of computing the validity of a formula in the initial state directly, for each subformula we use the appropriate subprocedure and compute the set of all states in which the subformula holds. We start with the smallest subformulas and then proceed to the larger ones, using the sets of states already computed. Let us now describe the algorithm more formally, including the aforementioned subprocedures.

The main procedure to check whether a given PCTL state formula Φ_0 holds for an MDP relies on the same concepts as for CTL. An iterative approach is used to compute the satisfaction sets $\mathsf{Sat}(\Phi) = \{s \in S \mid s \models \Phi\}$ of all subformulas Φ of Φ_0. The treatment of the propositional logic fragment of PCTL follows directly from the definition of the semantics. We will concentrate here on explaining how to deal with probabilistic features. The algorithms we give run in polynomial time if the cost bounds and cost functions are given in unary. Hence, checking whether a given formula holds can be done in polynomial time under these assumptions.

In the sequel, let $\mathscr{M} = (S, \mathsf{Act}, \mathsf{P}, s_{\mathsf{init}}, \mathsf{AP}, \mathsf{L}, \mathsf{C})$ be an MDP as in Sect. 28.2.2.

28.4.1 Probability Operator

Suppose that $\Phi = \mathbb{P}_{\sim p}(\varphi)$. We consider here the case of upper probability bounds, i.e., $\sim \in \{\leq, <\}$, so the task is to compute maximal probabilities of satisfying φ for every state. The set $\mathsf{Sat}(\Phi)$ can then be identified easily, as we have

$$\mathsf{Sat}(\Phi) = \left\{ s \in S \;\middle|\; \mathsf{Pr}_s^{\max}(\varphi) \sim p \right\}.$$

Lower probability bounds (i.e., the case when $\sim \in \{\geq, >\}$) can be treated similarly, but using minimum probability instead (see, e.g., [4, 99] for details). We distinguish three possible cases for the outermost operator of the path formula φ. For the proper state subformulas of φ, we can assume that the satisfaction sets $\mathsf{Sat}(\varphi)$ have already been computed. This allows us to treat them as atomic propositions.

First, we consider the **next-operator**. If $\varphi = \bigcirc \Psi$ then the maximal probabilities for φ are obtained by

$$\mathsf{Pr}_s^{\max}(\varphi) = \max_{\alpha \in \mathsf{Act}(s)} \mathsf{P}\big(s, \alpha, \mathsf{Sat}(\Psi)\big)$$

where $\mathsf{P}(s,\alpha,\mathsf{Sat}(\Psi)) = \sum_{t\in\mathsf{Sat}(\Psi)} \mathsf{P}(s,\alpha,t)$. An optimal simple scheduler simply assigns an action α to the state s that maximizes the value $\mathsf{P}(s,\alpha,\mathsf{Sat}(\Psi))$.

We now address the **until-operator** and suppose that $\varphi = \Phi_1 \mathsf{U} \Phi_2$. We first apply graph algorithms to compute the sets

$$S_0 = \left\{ s \in S \;\middle|\; \mathsf{Pr}_s^{\max}(\Phi_1 \mathsf{U} \Phi_2) = 0 \right\}$$
$$S_1 = \left\{ s \in S \;\middle|\; \mathsf{Pr}_s^{\max}(\Phi_1 \mathsf{U} \Phi_2) = 1 \right\}.$$

Note that S_0 is equal to the set $\{s \in S \mid \forall \pi \in \mathsf{Paths}(s) \mid \pi \not\models \Phi_1 \mathsf{U} \Phi_2\}$ which can be obtained using standard algorithms for non-probabilistic model checking (see Chap. 2). The set S_1 can be computed by iterating the following steps (1) and (2), where we start with the set of all states and keep pruning all actions and states that might lead to not satisfying the formula. Step (1) removes all states t from which no path satisfying $\Phi_1 \mathsf{U} \Phi_2$ starts. Step (2) considers all the remaining states s and removes all actions α from $\mathsf{Act}(s)$ such that $\mathsf{P}(s,\alpha,t) > 0$ for some state t that has been removed in step (1). The set of states that are not removed after repeating steps (1) and (2) constitutes the set S_1.

Let $S_? = S \setminus (S_0 \cup S_1)$ and $x_s = \mathsf{Pr}_s^{\max}(\Phi_1 \mathsf{U} \Phi_2)$ for $s \in S$. Clearly, $x_s = 0$ if $s \in S_0$, $x_s = 1$ if $s \in S_1$ and[2] $0 < x_s = \mathsf{Pr}_s^{\max}(\Phi_1 \mathsf{U} S_1) < 1$ if $s \in S_?$. The values x_s for $s \in S_?$ are obtained as the unique solution of the *linear program* [72] given by the inequalities

$$x_s \geq \sum_{t \in S_?} \mathsf{P}(s,\alpha,t) \cdot x_t + \mathsf{P}(s,\alpha,S_1) \quad \text{for all } \alpha \in \mathsf{Act}(s)$$

where $\sum_{s\in S_?} x_s$ is minimal and where $\mathsf{P}(s,\alpha,S_1) = \sum_{u\in S_1} \mathsf{P}(s,\alpha,u)$.

Intuitively, the inequalities of the above form capture the idea that the probability in state s must be at least the weighted sum of probabilities of the one-step successors, for any action α. Notice that every state is considered at most once in the sum, since $S_? \cap S_1 = \emptyset$.

A simple scheduler $\mathscr{U}$ with $\mathsf{Pr}_s^{\mathscr{U}}(\Phi_1 \mathsf{U} \Phi_2) = x_s = \mathsf{Pr}_s^{\max}(\Phi_1 \mathsf{U} \Phi_2)$ is obtained by carefully choosing, for each state $s \in S_1$, an action α with $\mathsf{P}(s,\alpha,S_1) = 1$ and, for each state $s \in S_?$, an action α that maximizes the value

$$\sum_{t \in S_?} \mathsf{P}(s,\alpha,t) \cdot x_t + \mathsf{P}(s,\alpha,S_1).^3$$

Some care is needed to ensure that the chosen action indeed makes some "progress" towards reaching a Φ_2-state. More formally, it is necessary to ensure that the actions taken will not avoid a Φ_2 state forever (the condition which captures this can be found in [4]). To illustrate the possible problem, consider the MDP from Fig. 3.

[2] The notation $\Phi_1 \mathsf{U} S_1$ is a shorthand for $\Phi_1 \mathsf{U} a$ where a is an atomic proposition satisfying $a \in L(s)$ if and only if $s \in S_1$.

[3] For the states $s \in S_0$ an arbitrary action can be chosen.

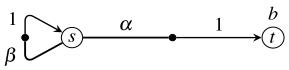


Fig. 3 An MDP showing that care needs to be taken when computing a scheduler $\mathscr{U}$ with $\mathsf{Pr}_s^{\mathscr{U}}(\Phi_1 \mathsf{U} \Phi_2) = \mathsf{Pr}_s^{\max}(\Phi_1 \mathsf{U} \Phi_2)$

Here, a simple scheduler that maximizes the probability for $\mathsf{tt}\,\mathsf{U}\,b$ must not take the action β for s, although $\mathsf{P}(s, \beta, S_1) = 1$ since $S_1 = \{s, t\}$.

Recall that all coefficients (transition probabilities in the MDP and the probability bound p) are rational, and hence the linear program above can be constructed in time polynomial in the size of $\mathscr{M}$. Because the linear program can be solved in polynomial time [72], the complexity of the problem to check whether an MDP satisfies a PCTL formula of the form $\mathbb{P}_{\leq p}(\Phi_1 \mathsf{U} \Phi_2)$ or $\mathbb{P}_{<p}(\Phi_1 \mathsf{U} \Phi_2)$ is also polynomial in the size of $\mathscr{M}$, assuming that the satisfaction sets for Φ_1 and Φ_2 are given.

Besides using well-known linear programming techniques to compute the vector $\vec{x} = (x_s)_{s \in S_?}$, one can use iterative approximation techniques. Most prominent are value and policy iteration, see, e.g., [99, 100].

In the *value iteration* approach, one starts with $x_s^{(0)} = 1$ for all $s \in S_1$ and $x_s^{(0)} = 0$ for all $s \in S_? \cup S_0$, and then successively computes

$$x_s^{(n+1)} \stackrel{\text{def}}{=} \max_{\alpha \in \mathsf{Act}(s)} \sum_{t \in S_?} \mathsf{P}(s, \alpha, t) \cdot x_t^{(n)} + \mathsf{P}(s, \alpha, S_1) \quad \text{for all } s \in S_?$$

until $\max_{s \in S_?} |x_s^{(n+1)} - x_s^{(n)}| < \varepsilon$ for some predefined tolerance $\varepsilon > 0$.

The idea of *policy iteration* is as follows. In each iteration, we select a simple scheduler $\mathscr{U}$ and compute the probabilities $\mathsf{Pr}_s^{\mathscr{U}}(\Phi_1 \mathsf{U} S_1)$ for $s \in S_?$ in the induced Markov chain (this can be done by solving a linear equation system). The method then "improves" the current simple scheduler $\mathscr{U}$ by searching for some state $s \in S_?$ such that

$$\mathsf{Pr}_s^{\mathscr{U}}(\Phi_1 \mathsf{U} S_1) < \max_{\alpha \in \mathsf{Act}(s)} \sum_{t \in S_?} \mathsf{P}(s, \alpha, t) \cdot \mathsf{Pr}_s^{\mathscr{U}}(\Phi_1 \mathsf{U} S_1) + \mathsf{P}(s, \alpha, S_1).$$

It then replaces $\mathscr{U}$ with $\mathscr{V}$ where $\mathscr{U}$ and $\mathscr{V}$ agree, except that $\mathscr{V}(s) = \alpha$ for some action $\alpha \in \mathsf{Act}(s)$ that maximizes $\sum_{t \in S_?} \mathsf{P}(s, \alpha, t) \cdot \mathsf{Pr}_s^{\mathscr{U}}(\Phi_1 \mathsf{U} S_1) + \mathsf{P}(s, \alpha, S_1)$. The next iteration is then performed with scheduler $\mathscr{V}$. If no improvement is possible, i.e., if

$$\mathsf{Pr}_s^{\mathscr{U}}(\Phi_1 \mathsf{U} S_1) = \max_{\alpha \in \mathsf{Act}(s)} \sum_{t \in S_?} \mathsf{P}(s, \alpha, t) \cdot \mathsf{Pr}_s^{\mathscr{U}}(\Phi_1 \mathsf{U} S_1) + \mathsf{P}(s, \alpha, S_1)$$

for all $s \in S_?$, then $\mathscr{U}$ maximizes the probability of $\Phi_1 \mathsf{U} \Phi_2$.

In practice, both value iteration and policy iteration outperform the linear-programming method, which does not scale to large models. The relative performance of value iteration and policy iteration varies by model, but the space and

time efficiency of value iteration can be easily improved so that it outperforms policy iteration. Interested readers are referred to [52] for a brief comparison.

It remains to explain the treatment of the **cost-bounded until-operator**. We consider here just the case of non-strict upper cost bounds. The task is to compute $\mathsf{Pr}_s^{\max}(\varphi)$ for all states $s \in S$, where $\varphi = \Phi_1 \mathsf{U}^{\leq c} \Phi_2$ and $c \in \mathbb{N}$. For $s \in S$ and $d \in \mathbb{N}$ we define

$$x_s(d) \stackrel{\text{def}}{=} \mathsf{Pr}_s^{\max}\big(\Phi_1 \mathsf{U}^{\leq d} \Phi_2\big).$$

Then, we have $x_s(d) = 1$ for each state $s \in \mathsf{Sat}(\Phi_2)$ and each cost bound $d \in \mathbb{N}$. Similarly, $x_s(d) = 0$ for each $d \in \mathbb{N}$ and state s satisfying $\mathsf{Pr}_s^{\max}(\Phi_1 \mathsf{U} \Phi_2) = 0$. Suppose now that $\mathsf{Pr}_s^{\max}(\Phi_1 \mathsf{U} \Phi_2) > 0$ and $s \not\models \Phi_2$. Thus, the recursive equations

$$x_s(d) = \max\left\{ \sum_{t \in S} \mathsf{P}(s,\alpha,t) \cdot x_t\big(d - \mathsf{C}(s,\alpha)\big) \;\middle|\; \alpha \in \mathsf{Act}(s), \mathsf{C}(s,\alpha) \leq d \right\}$$

hold true, where the maximum over the empty set is defined to be 0. That is, $x_s(d) = 0$ if $\mathsf{C}(s,\alpha) > d$ for all actions $\alpha \in \mathsf{Act}(s)$. Assuming that $\mathsf{C}(s,\alpha) > 0$ for all states s and enabled actions α, the above formulas for $x_s(d)$ can be computed by an iterative procedure, e.g., by employing a dynamic programming approach using the above equations. This yields the desired values $\mathsf{Pr}_s^{\max}(\varphi) = x_s(c)$. If $\mathsf{C}(s,\alpha) = 0$ for some states s and some actions $\alpha \in \mathsf{Act}(s)$ then the solution can be obtained as a solution to the linear program L_c which minimises $\sum_{s \in S} \sum_{0 \leq d \leq c} x_s(d)$, subject to

$$x_s(d) = 0 \quad \text{for } d < 0$$

$$x_s(d) = 1 \quad \text{for } d \geq 0 \text{ and } s \in \mathsf{Sat}(\Phi_2)$$

$$x_s(d) \geq \sum_{t \in S} P(s,\alpha,t) \cdot x_t\big(d - \mathsf{C}(s,\alpha)\big) \quad \text{for } d \geq 0,\ s \notin \mathsf{Sat}(\Phi_2) \text{ and } \alpha \in \mathsf{Act}(s)$$

where L_c contains variables $x_s(d)$ for $-M \leq d \leq c$ where M is the maximal number assigned by C. This approach can be optimised to consecutively solving $d+1$ linear programs $L'_0, \ldots, L'_c$, where $L'_0 = L_0$ and for $1 \leq i \leq c$ the linear program L'_i is obtained from L_i by turning the variables $x_s(j)$ for $j < i$ into constants whose values were already computed earlier.

28.4.2 Expectation Operator

Suppose now that the task is to compute the satisfaction set $\mathsf{Sat}(\mathbb{E}_{\sim c}(C))$, where C is the random variable $\mathsf{cost}[\cdot]$ associated with the reachability condition $\Diamond \Psi$, the total cost within the first k steps (i.e., C is "$\leq k$"), or the instantaneous cost incurred by the k-th step (i.e., C is "$=k$"). Again, we just consider the case of maximal expected cost where the goal is to compute $\mathsf{Ex}_s^{\max}(C)$ for all states s. The set $\mathsf{Sat}(\mathbb{E}_{\sim c}(C))$ is then obtained by collecting all states s where $\mathsf{Ex}_s^{\max}(C) \sim c$.

Let us first address the case of **cumulated cost within k steps**. We can rely on the iterative computation scheme

$$\mathsf{Ex}_s^{\max}\big(\mathsf{cost}[{\leq}n]\big) = \max_{\alpha \in \mathsf{Act}(s)} \left(\mathsf{C}(s,\alpha) + \sum_{t \in S} \mathsf{P}(s,\alpha,t) \cdot \mathsf{Ex}_t^{\max}\big(\mathsf{cost}[{\leq}n{-}1]\big) \right)$$

for $1 \leq n \leq k$ and $\mathsf{Ex}_s^{\max}(\mathsf{cost}[{\leq}0]) = 0$.

In the case of **instantaneous cost at time step** k, the equations have the form

$$\mathsf{Ex}_s^{\max}\big(\mathsf{cost}[{=}1]\big) = \max_{\alpha \in \mathsf{Act}(s)} \mathsf{C}(s,\alpha)$$

$$\mathsf{Ex}_s^{\max}\big(\mathsf{cost}[{=}n]\big) = \max_{\alpha \in \mathsf{Act}(s)} \sum_{t \in S} \mathsf{P}(s,\alpha,t) \cdot \mathsf{Ex}_t^{\max}\big(\mathsf{cost}[{=}n{-}1]\big)$$

for $1 < n \leq k$.

We now sketch the main steps for the computation of the **maximal expected cost for the reachability objective** $\Diamond\Psi$. We first apply techniques for the standard until-operator (see Sect. 28.3) to compute $\mathsf{Pr}_s^{\min}(\Diamond\Psi)$ for all states s in $\mathscr{M}$.

If t is a state in $\mathscr{M}$ with $\mathsf{Pr}_t^{\min}(\Diamond\Psi) < 1$ then there exists a scheduler $\mathscr{U}$ such that $\mathsf{Pr}_t^{\mathscr{U}}(\Diamond\Psi) < 1$. But then $\mathsf{Ex}_t^{\mathscr{U}}(\mathsf{cost}[\Diamond\Psi])$ is infinite, and therefore

$$\mathsf{Ex}_t^{\max}\big(\mathsf{cost}[\Diamond\Psi]\big) = \infty.$$

The remaining task is to compute $\mathsf{Ex}_s^{\max}(\mathsf{cost}[\Diamond\Psi])$ for all states $s \in S'$ where

$$S' = \big\{ s \in S \bigm| \mathsf{Pr}_s^{\min}(\Diamond\Psi) = 1 \big\}.$$

Note that, if $s \in S' \setminus \mathsf{Sat}(\Psi)$, then for all actions $\alpha \in \mathsf{Act}(s)$ and all states u with $\mathsf{P}(s,\alpha,u) > 0$ we have $u \in S'$. The enabled actions of the states $s \in \mathsf{Sat}(\Psi)$ are irrelevant. We may suppose that for these s, $\mathsf{Act}(s)$ is a singleton set $\{\alpha\}$ with $\mathsf{P}(s,\alpha,s) = 1$. Clearly, for $s \in \mathsf{Sat}(\Psi)$ we have $\mathsf{Ex}_s^{\max}(\mathsf{cost}[\Diamond\Psi]) = 0$. For all other states $s \in S' \setminus \mathsf{Sat}(\Psi)$, we have

$$\mathsf{Ex}_s^{\max}\big(\mathsf{cost}[\Diamond\Psi]\big) = \max_{\alpha \in \mathsf{Act}(s)} \left(\mathsf{C}(s,\alpha) + \sum_{u \in S'} \mathsf{P}(s,\alpha,u) \cdot \mathsf{Ex}_u^{\max}\big(\mathsf{cost}[\Diamond\Psi]\big) \right).$$

These values can again be computed using linear programming techniques or the value or policy iteration schemes.

Example 5 Consider the MDP from Example 1 and the formula $\mathbb{E}_{\leq 5}(\leq 4)$. For all $0 \leq i \leq 4$, let x^i denote the tuple

$$\big(\mathsf{Ex}_{s_0}^{\max}\big(\mathsf{cost}[{\leq}i]\big), \mathsf{Ex}_{s_1}^{\max}\big(\mathsf{cost}[{\leq}i]\big), \mathsf{Ex}_{s_2}^{\max}\big(\mathsf{cost}[{\leq}i]\big), \mathsf{Ex}_{s_3}^{\max}\big(\mathsf{cost}[{\leq}i]\big)\big).$$

We iteratively compute the following tuples by applying value iteration

$$\begin{aligned} x^1 &= (\quad 1, \quad 4, 0, 0\ ) \\ x^2 &= (\quad 5, \quad 4, 0, 0\ ) \\ x^3 &= (\quad 5, 4.5, 0, 0\ ) \\ x^4 &= (\ 5.5, 4.5, 0, 0\ ) \end{aligned}$$

and we conclude that the formula $\mathbb{E}_{\leq 5}(\leq 4)$ is not satisfied, because the maximal cumulated cost in s_0 is 5.5.

Next, consider again the same MDP, but this time together with the formula $\mathbb{P}_{\leq 0.5}(\neg \mathit{init}\, \mathsf{U}\, \mathit{succ})$, and suppose we want to know precisely the states in which the formula holds. We start by parsing the formula from the smallest subformulas. The subformula *init* is satisfied in s_0, and *succ* in s_2. Further, the subformula $\neg \mathit{init}$ is satisfied in the states s_1, s_2, and s_3. A more demanding task is to compute $\mathsf{Pr}_s^{\max}(\neg \mathit{init}\, \mathsf{U}\, \mathit{succ})$. We compute the sets S_0 and S_1, which are

$$S_0 = \{s_0, s_3\} \quad \text{and} \quad S_1 = \{s_2\}.$$

This leaves us with the set $S_? = \{s_1\}$. We construct the following simple linear program

$$\begin{aligned} &\text{minimize } x_{s_1} \text{ subject to} \\ &x_{s_1} \geq x_{s_1} \\ &x_{s_1} \geq 0.3. \end{aligned}$$

The solution to the above program is $x_{s_1} = 0.3$, and hence we can conclude that the formula $\mathbb{P}_{\leq 0.5}(\neg \mathit{init}\, \mathsf{U}\, \mathit{succ})$ holds in states s_0, s_1 and s_3.

28.5 Linear Temporal Logic

We continue this chapter with a brief overview of model checking Markov decision processes against properties expressed in linear temporal logic (LTL). In this section we define the logic and in the next section we show how the model-checking algorithm works. The logic LTL that we will use is standard, as defined in Chap. 2, except that we use only a subset of LTL which does not allow us to reason about the past, and whose predicates are actions of an MDP. Having predicates over actions and not over states is only a matter of convention; all the constructions and algorithms we present here can be easily modified to work with state predicates.

28.5.1 Syntax of LTL

For the purposes of this chapter, the syntax of LTL is as follows,

$$\varphi ::= \mathsf{tt} \mid \alpha \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \bigcirc \varphi \mid \varphi_1 \mathsf{U} \varphi_2$$

where tt stands for the constant truth value "true", and α is an action, i.e., an element of the set of actions Act. We write U to denote the *until*-operator, instead of $\mathscr{U}$ used in Chap. 2.

28.5.2 Semantics of LTL

The semantics of our logic LTL is defined on *traces* of paths of an MDP. A trace for an infinite path $\pi = s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \xrightarrow{\alpha_3} \dots$ is the infinite word $\mathsf{trace}(\pi) = \alpha_1\alpha_2\alpha_3\dots$ of actions. Let $w = \alpha_0\alpha_1\dots$ be an infinite word over the alphabet of actions Act, and let $w{\uparrow}^n$ denote the suffix of w starting with α_n. Then,

$$
\begin{array}{lll}
w \models \mathsf{tt} & & \\
w \models \alpha & \text{iff} & \alpha = \alpha_0 \\
w \models \neg\phi & \text{iff} & w \not\models \phi \\
w \models \varphi_1 \wedge \varphi_2 & \text{iff} & w \models \varphi_1 \text{ and } w \models \varphi_2 \\
w \models \bigcirc\varphi & \text{iff} & w{\uparrow}^1 \models \varphi \\
w \models \varphi_1 \mathsf{U} \varphi_2 & \text{iff} & \text{there exists } k \in \mathbb{N} \text{ with } w{\uparrow}^k \models \varphi_2 \text{ and } w{\uparrow}^i \models \varphi_1 \text{ for } 0 \leq i < k.
\end{array}
$$

As in the case of PCTL, it can be shown that the set of all infinite paths that satisfy a given LTL formula is always measurable.

28.5.3 Derived Operators

Similarly to PCTL, we can define Boolean operators such as ff, $\vee$ and $\rightarrow$ from negation and conjunction, for example

$$\varphi_1 \vee \varphi_2 \stackrel{\text{def}}{=} \neg(\neg\varphi_1 \wedge \neg\varphi_2) \quad \text{and} \quad \varphi_1 \rightarrow \varphi_2 \stackrel{\text{def}}{=} (\neg\varphi_1) \vee \varphi_2.$$

The eventually-operator $\Diamond$ and the always-operator $\Box$ are obtained by

$$\Diamond\varphi \stackrel{\text{def}}{=} \mathsf{tt}\,\mathsf{U}\varphi \quad \text{and} \quad \Box\varphi \stackrel{\text{def}}{=} \neg\Diamond\neg\varphi.$$

For simplicity, we did not introduce a cost-bounded version of the until-operator $\mathsf{U}^{\sim c}$, but in principle there is nothing preventing us from doing so. We point out that the notation would become cumbersome; in particular, the definition of the Rabin automaton below would then need to take costs of state-action pairs into consideration.

28.5.4 LTL Model-Checking Problem

Let $\mathscr{M} = (S, \mathsf{Act}, \mathsf{P}, s_{\mathsf{init}}, \mathsf{AP}, \mathsf{L}, \mathsf{C})$ be an MDP and $\mathbb{P}_{\sim p}(\varphi)$ an *LTL state property*, where $\sim$ is a comparison operator $\leq$ or $<$, $p \in [0,1] \cap \mathbb{Q}$ and φ is an LTL formula.

The *LTL model-checking problem* is to decide whether

$$\Pr^{\max}_{s_{\text{init}}}\{\pi \in \text{Paths} \mid \text{trace}(\pi) \models \varphi\} \sim p.$$

We can define the model-checking problem similarly for the comparison operators $\geq$ or $>$; in that case we ask whether

$$\Pr^{\min}_{s_{\text{init}}}\{\pi \in \text{Paths} \mid \text{trace}(\pi) \models \varphi\} \sim p.$$

Because the LTL formulas are closed under negation, we have

$$\begin{aligned}
&\Pr^{\min}_{s_{\text{init}}}(\{\pi \in \text{Paths} \mid \text{trace}(\pi) \models \varphi\}) \\
&\quad = 1 - \Pr^{\max}_{s_{\text{init}}}(\{\pi \in \text{Paths} \mid \text{trace}(\pi) \not\models \varphi\}) \\
&\quad = 1 - \Pr^{\max}_{s_{\text{init}}}(\{\pi \in \text{Paths} \mid \text{trace}(\pi) \models \neg\varphi\})
\end{aligned}$$

and so without loss of generality we can restrict our interest to the case of computing maximal probabilities.

28.6 Model-Checking Algorithms for MDPs and LTL

In this section we describe a model-checking algorithm for MDPs and LTL. Before going into formal definitions, let us describe it informally. We solve the LTL model-checking problem using ω-regular automata. Every LTL formula can be transformed into an automaton which accepts exactly the words on which the formula holds. We then build the *product* of the MDP and the automaton, and show that the problem of computing the optimal probability with which the automaton accepts traces of the MDP is equal to the problem of computing the optimal probability of reaching certain states in the product. The latter can be computed using algorithms from previous sections. The reader may observe that the outline of the algorithm is similar to the (non-probabilistic) LTL model-checking algorithm from Chap. 2. The major difference is that, instead of looking for one path in the product (called synchronous composition in Chap. 2), we need to determine the probability of certain paths. It turns out that, for this purpose, the definition of a just discrete system is not sufficient. The solution we present here uses Rabin automata, whose crucial property is that it has no nondeterminism.

The algorithm runs in time polynomial in the size of the MDP and doubly-exponential in the size of the LTL formula. From the complexity-theoretic point of view, the complexity bound is optimal since the model-checking problem for Markov decision processes and LTL state properties is known to be complete for the complexity class 2EXPTIME, even for qualitative LTL state properties [40].

Let us now describe the algorithm formally. We begin by introducing the notion of *deterministic Rabin automata* and stating that, for every LTL formula φ, there is a deterministic Rabin automaton that accepts exactly the set of words satisfying φ.

Definition 1 (Deterministic Rabin Automaton (DRA)) A *deterministic Rabin automaton* is a tuple $\mathscr{A} = (Q, \mathsf{Act}, \delta, q_{init}, Acc)$, where Q is a finite set of states, $q_{init} \in Q$ is an initial state, Act is a finite input alphabet, $\delta : Q \times \mathsf{Act} \to Q$ is a transition function, and $Acc = \{(L_1, K_1), (L_2, K_2), \ldots, (L_k, K_k)\}$, for $k \in \mathbb{N}$ and $L_i, K_i \subseteq Q$, $1 \leq i \leq k$, is a set of accepting tuples of states.

We do not study Rabin automata in detail here and only mention their properties directly relevant to LTL model checking. We refer the reader to Chap. 4 or to [56] for additional details.

Let $\mathscr{A} = (Q, \mathsf{Act}, \delta, q_{init}, Acc)$ be a DRA. For every infinite word $w = \alpha_0\alpha_1\alpha_2\ldots$ over the input alphabet Act there is a unique sequence $q_0\alpha_0 q_1\alpha_1 q_2\alpha_2\cdots$ where $q_0 = q_{init}$, and $\delta(q_i, \alpha_i) = q_{i+1}$ for all $i \geq 0$. The word w is *accepted* by $\mathscr{A}$ if there is $(L, K) \in Acc$ such that $q_i \in L$ for only finitely many i, and $q_j \in K$ for infinitely many j. The set of all infinite words over Act that $\mathscr{A}$ accepts is called the *language* of $\mathscr{A}$ and is denoted $\mathscr{L}(A)$.

As mentioned above, for every LTL formula φ we can construct a DRA $\mathscr{A}_\varphi$ with the input alphabet Act such that for all $w = \alpha_1\alpha_2\ldots$ we have

$$w \models \varphi \quad \Longleftrightarrow \quad w \in \mathscr{L}(\mathscr{A}_\varphi).$$

The construction of $\mathscr{A}_\varphi$ is non-trivial and we do not present it in this chapter, referring the reader to [14, 41, 107]. Note that, in general, the size of $\mathscr{A}_\varphi$ can be up to doubly exponential in the size of φ. In practice, however, this is often not a serious problem since LTL formulas expressing useful properties tend to be small compared to the size of the MDP.

Having defined the DRA $\mathscr{A}_\varphi$, we reduce the problem of computing the maximal probability of paths satisfying φ in $\mathscr{M}$ to the problem of reaching a certain set of states in a *product* MDP. The product MDP is defined so that its behaviour mimics that of the original MDP, but in addition it remembers the state of the automaton in which it ends after reading the sequence of actions performed so far.

Definition 2 (Product of an MDP and a DRA) Let $\mathscr{M} = (S, \mathsf{Act}, \mathsf{P}, s_{\mathsf{init}}, \mathsf{AP}, \mathsf{L}, \mathsf{C})$ be an MDP and $\mathscr{A} = (Q, \mathsf{Act}, \delta, q_{init}, Acc)$ a DRA. Their *product* $\mathscr{M} \otimes \mathscr{A}$ is the MDP $(S \times Q, \mathsf{Act}, \mathsf{P}', (s_{\mathsf{init}}, q_{init}), \mathsf{AP}, \mathsf{L}', \mathsf{C}')$ where for any $(s, q) \in S \times Q$ and $\alpha \in \mathsf{Act}$ we define

$$\mathsf{P}'\big((s,q),\alpha,(s',q')\big) = \begin{cases} \mathsf{P}(s,\alpha,s') & \text{if } \delta(q,\alpha) = q' \\ 0 & \text{otherwise.} \end{cases}$$

The elements L' and C' are defined arbitrarily.

A path $(s_0, q_0) \xrightarrow{\alpha_1} (s_1, q_1) \xrightarrow{\alpha_2} (s_2, q_2) \xrightarrow{\alpha_3} \ldots$ in a product MDP is accepting if there is $(L, K) \in Acc$ such that $q_i \in L$ for only finitely many i and $q_j \in K$ for infinitely many j.

It can be proved that, for every state s and scheduler $\mathscr{U}$ in $\mathscr{M}$, there is a scheduler $\mathscr{V}$ in $\mathscr{M} \otimes \mathscr{A}$ such that

$$\begin{aligned}&\mathsf{Pr}^{\mathscr{M},\mathscr{U}}\left(\left\{\pi \in \mathsf{Paths}^{\mathscr{M}}(s) \;\middle|\; \mathsf{trace}(\pi) \in \mathscr{L}(\mathscr{A})\right\}\right)\\ &\quad = \mathsf{Pr}^{\mathscr{M}\otimes\mathscr{A},\mathscr{V}}\left(\left\{\pi \in \mathsf{Paths}^{M\otimes\mathscr{A}}\left((s, q_{init})\right) \;\middle|\; \pi \text{ is an accepting path}\right\}\right).\end{aligned}$$

This is essentially because the product only extends the original by keeping track of a computation of a DRA, and does not alter the power of schedulers.

So far, we have reduced the problem of LTL model checking to the problem of determining the maximal probability of accepting paths in a product MDP. To determine this probability, we introduce the notion of accepting end components, which identify the states for which there is a scheduler ensuring that almost all paths starting in these states are accepting. An *accepting end component* (EC) of $\mathscr{M} \otimes \mathscr{A}$ is a pair $(\bar{S}, \bar{\mathsf{P}})$ comprising a subset $\bar{S} \subseteq S \times Q$ of states and partial transition probability function $\bar{\mathsf{P}} : \bar{S} \times \mathsf{Act} \times \bar{S} \to [0, 1] \cap \mathbb{Q}$ satisfying the following conditions:

1. $(\bar{S}, \bar{\mathsf{P}})$ determines a sub-MDP of $\mathscr{M} \otimes \mathscr{A}$, i.e., for all $s' \in \bar{S}$ and $\alpha \in \mathsf{Act}$ we have $\sum_{s'' \in \bar{S}} \bar{\mathsf{P}}(s', \alpha, s'') = 1$, and, if $\bar{\mathsf{P}}(s', \alpha, s'')$ is defined, then $\bar{\mathsf{P}}(s', \alpha, s'') = \mathsf{P}'(s', \alpha, s'')$;
2. the underlying graph of $(\bar{S}, \bar{\mathsf{P}})$ is strongly connected;
3. there is $(L, K) \in Acc$ such that:
   a. all $(s, q) \in \bar{S}$ satisfy $q \notin L$;
   b. there is $(s, q) \in \bar{S}$ satisfying $q \in K$.

Using the above condition for an accepting path, together with the property that, once an end component is entered, all its states can be visited infinitely often almost surely [4], we can further show the following. Let $T \subseteq S \times Q$ such that $(s', q') \in T$ if and only if (s', q') appears in some accepting end component of $\mathscr{M} \otimes \mathscr{A}$, then we have

$$\begin{aligned}&\mathsf{Pr}^{\max}_{s}\left\{\pi \in \mathsf{Paths}^{\mathscr{M}}(s) \;\middle|\; \mathsf{trace}(\pi) \in \mathscr{L}(\mathscr{A})\right\}\\ &\quad = \mathsf{Pr}^{\max}_{(s,q_{init})}\left(\left\{\pi \in \mathsf{Paths}^{\mathscr{M}\otimes\mathscr{A}}\left((s, q_{init})\right) \;\middle|\; \pi \text{ contains a state from } T\right\}\right).\end{aligned}$$

Thus, we have reduced model checking of LTL properties to (i) the computation of accepting end components in $\mathscr{M} \otimes \mathscr{A}_\varphi$, and (ii) the computation of maximum probabilities of reaching these end components. The second step is a special case of the problems studied in Sect. 28.4. The first step can be done efficiently using the results of [4, 14]; an approach which is simpler to comprehend, but less efficient, is to use PCTL model checking to identify all the states that lie in an accepting component and satisfy the condition 3b. above. These are exactly the states (s, q) for which there is $(L, K) \in Acc$ such that $q \in K$ and it is possible to return to (s, q) with probability 1, passing only through states (s', q') with $q' \notin L$. A state (s, q) satisfies this condition if and only if it satisfies a formula $\neg\mathbb{P}_{<1}(\bigcirc\neg\mathbb{P}_{<1}\, p_{\neg L} \,\mathsf{U}\, p_{(s,q)})$ for some $(L, K) \in Acc$ with $q \in K$, where $p_{(s,q)}$ holds only in (s, q) and $p_{\neg L}$ holds in all states (s', q') with $q' \notin L$. In step (ii) it is then sufficient to maximise the probability of reaching such states.

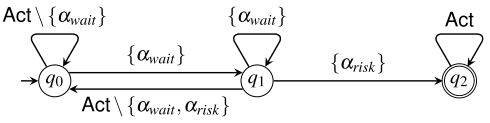


Fig. 4 A DRA for the formula $\Diamond(\alpha_{wait} \wedge \bigcirc\alpha_{risk})$

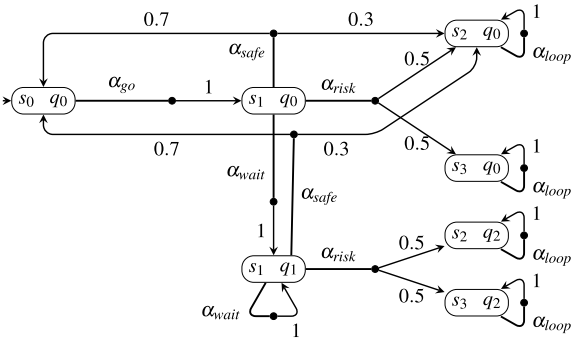


Fig. 5 The product MDP $\mathscr{M} \otimes \mathscr{A}$ for Example 6

Example 6 Consider the MDP from Example 1 together with the formula $\Phi = \Diamond(\alpha_{wait} \wedge \bigcirc\alpha_{risk})$, and assume we want to compute the maximal probability of satisfying this formula. We follow the procedure described above and first convert Φ to an equivalent DRA $\mathscr{A} = (Q, \mathsf{Act}, \delta, q_{init}, Acc)$. Using one of the cited methods, we might, for example, obtain the automaton shown in Fig. 4. Here, $Q = \{q_0, q_1, q_2\}$, $q_{init} = q_0$, $\delta(q, \alpha) = q'$ whenever there is an arrow from q to q' labelled with a set containing α, and $Acc = \{(\emptyset, \{q_2\})\}$.

Next, we construct the product of $\mathscr{M}$ and $\mathscr{A}$, yielding the MDP $\mathscr{M} \otimes \mathscr{A}$ from Fig. 5 (note that only the states reachable from the initial state (s_0, q_0) are drawn). The MDP $\mathscr{M} \otimes \mathscr{A}$ contains two accepting end components, one containing the state (s_2, q_2) and a self-loop, and the other containing the state (s_3, q_2) and a self-loop.

It is now easy to apply the algorithms from Sect. 28.4 and calculate that the maximal probability of reaching one of these end components from the initial state is equal to 1.

28.7 Tools, Applications and Model Construction

28.7.1 Tool Support

There are several software tools which implement probabilistic model checking for Markov decision processes. One of the most widely used is PRISM [81], an open-source tool available from [98] which supports both PCTL and LTL model checking as described here, including the probabilistic and expectation operators. PRISM uses a probabilistic variant of reactive modules as a modelling notation, and additionally supports model checking for discrete- and continuous-time Markov chains and probabilistic timed automata. The tools LIQUOR [38] and ProbDiVinE [16] implement

LTL model checking for MDPs: LIQUOR uses Probmela, which is a variant of the SPIN Promela modelling language, whereas ProbDiVinE provides a parallel implementation. RAPTURE [70] and PASS [58] apply abstraction-refinement techniques.

A key challenge when implementing the algorithms is the state-explosion problem, well known from other fields of model checking, and also discussed in Chap. 8 of this book. Different tools take a different approach to overcome this problem. The tool PRISM, for example, mainly uses a symbolic approach (see [6, 9] or Chap. 31) and instead of storing the state space explicitly it stores it using a variant of *binary decision diagrams* [54]. ProbDiVinE makes use of *distributed model checking*, while LIQUOR applies *partial-order reduction techniques* (see Chap. 6) to reduce the state space. Several other methods to tackle the state-explosion problem have been proposed for probabilistic model checking, including *symmetry reduction* [44, 78], game-based *quantitative abstraction refinement* [74, 80], compositional probabilistic verification [42, 51, 82, 83], or algorithms for *simulation* and *bisimulation* relations [31, 110]. Techniques to improve efficiency of probabilistic model checking include *approximate probabilistic model-checking* [88], *statistical model checking* [19, 25, 89, 108, 109] and *incremental verification* [86].

28.7.2 Applications

Probability is pervasive, and Markov decision processes underpin modelling and analysis of a wide range of applications [99]. Probabilistic model checking, and PRISM in particular, has been successfully applied to analyse and in some cases detect flaws in a variety of application domains, including analysis of communication, security, privacy and anonymity protocols, efficiency of power management protocols, correctness of randomised coordination algorithms, performance of computer systems and nanotechnology designs, *in silico* exploration of biological signalling, detecting design flaws in DNA circuits, analysis of spread of diseases, scheduling, planning, and controller synthesis (see, e.g., [45, 62, 79, 94]). More case studies are available at the PRISM tool website [98].

28.7.3 Construction of Probabilistic Models

The usefulness and precision of the results obtained by the probabilistic model-checking techniques presented here crucially depend on the adequacy of the model, and in particular on the probability values. Several methods have been proposed in the literature that support the stepwise and compositional design of probabilistic models for systems with many components, ranging from approaches that use stochastic process algebras (see, e.g., [3, 71]), probabilistic variants of Petri nets (see, e.g., [2]), or bespoke models (see, e.g., [5]) to high-level modelling languages

with guarded commands, probabilistic choice, and imperative programming language concepts [8, 20, 60, 66, 73]. Such approaches can indeed be very helpful when constructing reasonable models that reflect the architectural structure of the system to be analysed, the control flow of its components, the interaction mechanism, and dependencies between components where the probabilities are known or given, as is the case in randomised protocols. However, such formal modelling approaches do not support the choice of the probability values. Estimating probability distributions is one of the core problems studied in statistics. Indeed, for many application areas, well-engineered statistical methods are available to derive good estimates for the probability values in the models used for the quantitative analysis. But even without the application of advanced statistical methods, probabilistic model-checking techniques can yield useful information on the quantitative system behavior. Repeated application of probabilistic model-checking techniques on models that only differ in the probability values might give insights into the significance or irrelevance of certain probabilistic parameters. The model of Markov decision processes also permits the representation of incomplete information on the probability values by nondeterministic choices between several probabilistic distributions. The results obtained by probabilistic model checking are lower or upper bounds for all models that result by resolving the nondeterministic choices using any convex combination of the chosen distributions. Alternatively, there are also methods that deal with probability intervals rather than specific probability values, and methods that operate with parametrized probabilistic models, see, e.g., [37, 43, 57, 103].

28.8 Extensions of the Model and Specification Notations

There are various models that extend Markov decision processes, such as *stochastic games* [33, 34, 36], in which there are two kinds of nondeterminism (sometimes called "angelic" and "demonic" nondeterminism), or *probabilistic timed automata* [84, 95], which extend timed automata as defined in Chap. 29 and allow for reasoning about time by adding real-time constraints on actions. Another class of related models are *continuous-time Markov Chains* and *continuous-time Markov decision processes* [99] in which we add a notion of time into the system and assume that the steps from one state to another are taken with delays governed by an exponential probability distribution. Continuous-time Markov Chains find applications, for example, in biochemistry (see, e.g., [29, 30, 39, 63, 92]). Note that MDPs as defined in this chapter are sometimes called *discrete-time* MDPs to reflect the intuition that each of their steps takes exactly 1 time unit. Also note that adding an exponential distribution on time makes it more difficult to define parallel composition of two systems, leading to an alternative model of *interactive Markov chains* (see, e.g., [28, 64]).

Probabilistic models with infinite state space have also been studied, where examples include models generated by pushdown systems (see, e.g., [22, 26, 49]) or lossy channel systems [1, 10, 69].

Recently [32], alternatives to deterministic Rabin automata, such as generalized Rabin automata [47, 76], have been shown suitable for probabilistic model checking. These automata can be smaller by orders of magnitude and thus induce a smaller product to be analyzed. See [18] for an overview of available tools for conversion of LTL to different types of Rabin automata and their performance.

The logics LTL and PCTL can be naturally combined into the logic PCTL* [17], which is itself a probabilistic variant of the logic CTL* [46]. There are also numerous reward-based properties not included in our definition of PCTL, for example a discounted reward or long-run average reward [4, 99]. There also exist different logics that allow us to reason about probabilities, one example being the works [67, 91, 93] which study a probabilistic variant of μ-calculus (see Chap. 26). A new direction started recently concerns studying multi-objective model-checking problems for Markov decision processes [23, 35, 48, 53].

A related problem is that of *controller synthesis*, where the question is whether there *exists* a satisfying scheduler (as opposed to the model-checking problem, where we ask whether *all* schedulers satisfy the formula). For the unrestricted controller-synthesis problem, an alternative semantics of PCTL has been studied [12, 24, 27], yielding undecidability results.

28.9 Conclusion

In this chapter, we have given an overview of probabilistic model checking, focusing on Markov decision processes as an operational model for nondeterministic-probabilistic systems against specifications given in temporal logics PCTL and LTL. The PCTL model-checking algorithm is similar to that for the logic CTL, where the parse tree of the formula is traversed bottom up and each subformula is treated separately. Model checking for the probabilistic and expectation operator reduces to a linear programming problem, which can be solved using a variety of methods.

In the case of LTL, we first translate the LTL formula into an equivalent deterministic Rabin automaton, and then reduce the model-checking problem to the problem of calculating the probability of reaching accepting end components in a product of the MDP and the automaton. The construction of a deterministic Rabin automaton for a given LTL formula can cause a doubly exponential blowup.

We have also presented a brief summary of tools that implement and extend the algorithms presented in this chapter, and listed various related formalisms that exist in the area of probabilistic model checking.

References

1. Abdulla, P., Baier, C., Iyer, P., Jonsson, B.: Reasoning about probabilistic lossy channel systems. In: Palamidessi, C. (ed.) Proc. CONCUR'00. LNCS, vol. 1877, pp. 320–330. Springer, Heidelberg (2000)

2. Ajmone-Marsan, M., Balbo, G., Conte, G., Donatelli, S., Franceschinis, G.: Modelling with Generalized Stochastic Petri Nets. Wiley Series in Parallel Computing. Wiley, New York (1995)
3. Aldini, A., Bernardo, M., Corradini, F.: A Process Algebraic Approach to Software Architecture Design. Springer, Heidelberg (2010)
4. de Alfaro, L.: Formal verification of probabilistic systems. Ph.D. thesis, Stanford University, Department of Computer Science (1997)
5. de Alfaro, L., Henzinger, T.A., Jhala, R.: Compositional methods for probabilistic systems. In: Larsen, K.G., Nielsen, M. (eds.) CONCUR. LNCS, vol. 2154, pp. 351–365. Springer, Heidelberg (2001)
6. de Alfaro, L., Kwiatkowska, M., Norman, G., Parker, D., Segala, R.: Symbolic model checking of probabilistic processes using MTBDDs and the Kronecker representation. In: Graf, S., Schwartzbach, M.I. (eds.) Proc. Tools and Algorithms for Construction and Analysis of Systems (TACAS). LNCS, vol. 1785, pp. 395–410. Springer, Heidelberg (2000)
7. Ash, R., Doléans-Dade, C.: Probability and Measure Theory. Harcourt/Academic Press, San Diego (2000)
8. Baier, C., Ciesinski, F., Größer, M.: ProbMeLa: a modeling language for communicating probabilistic systems. In: Proc. of the 2nd ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE), pp. 57–66. IEEE, Piscataway (2004)
9. Baier, C., Clarke, E., Hartonas-Garmhausen, V., Kwiatkowska, M., Ryan, M.: Symbolic model checking for probabilistic processes. In: Degano, P., Gorrieri, R., Marchetti-Spaccamela, A. (eds.) Proc. International Colloqium on Automata, Languages and Programming (ICALP). LNCS, vol. 1256, pp. 430–440. Springer, Heidelberg (1997)
10. Baier, C., Engelen, B.: Establishing qualitative properties for probabilistic lossy channel systems: an algorithmic approach. In: Katoen, J.-P. (ed.) Intl. AMAST Workshop, ARTS. LNCS, vol. 1601, pp. 34–52. Springer, Heidelberg (1999)
11. Baier, C., Größer, M., Ciesinski, F.: Model checking linear time properties of probabilistic systems. In: Droste, M., Kuich, W., Vogler, H. (eds.) Handbook of Weighted Automata, Monographs in Theoretical Computer Science. An EATCS Series, pp. 519–570. Springer, Heidelberg (2009)
12. Baier, C., Größer, M., Leucker, M., Bollig, B., Ciesinski, F.: Controller synthesis for probabilistic systems. In: Lévy, J.J., Mayr, E., Mitchell, J. (eds.) Proc. 3rd IFIP Int. Conf. Theoretical Computer Science (TCS'06), pp. 493–5062. Kluwer Academic, Dordrecht (2004)
13. Baier, C., Haverkort, B.R., Hermanns, H., Katoen, J.P.: Performance evaluation and model checking join forces. Commun. ACM **53**(9), 76–85 (2010)
14. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press, Cambridge (2008)
15. Baier, C., Kwiatkowska, M.: Model checking for a probabilistic branching time logic with fairness. Distrib. Comput. **11**, 125–155 (1998)
16. Barnat, J., Brim, L., Černá, I., Češka, M., Tůmová, J.: ProbDiVinE-MC: multi-core LTL model checker for probabilistic systems. In: Proceedings of the 2008 Fifth International Conference on Quantitative Evaluation of Systems, pp. 77–78. IEEE, Washington (2008)
17. Bianco, A., De Alfaro, L.: Model checking of probabilistic and non-deterministic systems. In: Thiagarajan, P.S. (ed.) Proceedings of Foundations of Software Technology and Theoretical Computer Science. LNCS, vol. 1026, pp. 499–513. Springer, Heidelberg (1995)
18. Blahoudek, F., Kretínský, M., Strejcek, J.: Comparison of LTL to deterministic Rabin automata translators. In: McMillan, K.L., Middeldorp, A., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning—Proceedings of the 19th International Conference, LPAR-19, Stellenbosch, South Africa, December 14–19, 2013. LNCS, vol. 8312, pp. 164–172. Springer, Heidelberg (2013)
19. Bogdoll, J., Fioriti, L.M.F., Hartmanns, A., Hermanns, H.: Partial order methods for statistical model checking and simulation. In: Bruni, R., Dingel, J. (eds.) FMOODS/FORTE. LNCS, vol. 6722, pp. 59–74. Springer, Heidelberg (2011)

20. Bohnenkamp, H., D'Argenio, P., Hermanns, H., Katoen, J.P.: MODEST: a compositional modeling formalism for hard and softly timed systems. IEEE Trans. Softw. Eng. **32**(10), 812–830 (2006)
21. Bolch, G., Greiner, S., de Meer, H., Trivedi, K.: Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications. Wiley-Interscience, New York (1998)
22. Brázdil, T.: Verification of probabilistic recursive sequential programs. Ph.D. thesis, Masaryk University (2007)
23. Brázdil, T., Brožek, V., Chatterjee, K., Forejt, V., Kučera, A.: Two views on multiple mean-payoff objectives in Markov decision processes. In: Proceedings of LICS'11, pp. 33–42. IEEE, Piscataway (2011)
24. Brázdil, T., Brožek, V., Forejt, V., Kučera, A.: Stochastic games with branching-time winning objectives. In: 21th IEEE Symp. Logic in Computer Science (LICS 2006), pp. 349–358. IEEE, Piscataway (2006)
25. Brázdil, T., Chatterjee, K., Chmelík, M., Forejt, V., Křetínský, J., Kwiatkowska, M., Parker, D., Ujma, M.: Verification of Markov decision processes using learning algorithms. In: Cassez, F., Raskin, J. (eds.) Proc. 12th International Symposium on Automated Technology for Verification and Analysis (ATVA'14). LNCS, vol. 8837, pp. 98–114. Springer, Heidelberg (2014)
26. Brázdil, T., Esparza, J., Kiefer, S., Kučera, A.: Analyzing probabilistic pushdown automata. Form. Methods Syst. Des. **43**(2), 124–163 (2013)
27. Brázdil, T., Forejt, V., Kučera, A.: Controller synthesis and verification for Markov decision processes with qualitative branching time objectives. In: Aceto, L., Damgård, I., Goldberg, L., Halldórsson, M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) Proc. 35th Int. Colloq. Automata, Languages and Programming, Part II (ICALP'08). LNCS, vol. 5126, pp. 148–159. Springer, Heidelberg (2008)
28. Brázdil, T., Hermanns, H., Krčál, J., Křetínský, J., Řehák, V.: Verification of open interactive Markov chains. In: D'Souza, D., Kavitha, T., Radhakrishnan, J. (eds.) FSTTCS. LIPIcs, vol. 18, pp. 474–485. Schloss Dagstuhl—Leibniz-Zentrum fuer Informatik, Dagstuhl (2012)
29. Brim, L., Češka, M., Dražan, S., Šafránek, D.: Exploring parameter space of stochastic biochemical systems using quantitative model checking. In: Sharygina and Veith [104], pp. 107–123
30. Cardelli, L.: Artificial biochemistry. In: Condon, A., Harel, D., Kok, J.N., Salomaa, A., Winfree, E. (eds.) Algorithmic Bioprocesses. Natural Computing Series, pp. 429–462. Springer, Heidelberg (2009)
31. Cattani, S., Segala, R.: Decision algorithms for probabilistic bisimulation. In: Brim, L., Jančar, P., Křetínský, M., Kučera, A. (eds.) Proc. 14th Int. Conf. Concurrency Theory (CONCUR'02). LNCS, vol. 2421, pp. 371–385. Springer, Heidelberg (2002)
32. Chatterjee, K., Gaiser, A., Křetínský, J.: Automata with generalized Rabin pairs for probabilistic model checking and LTL synthesis. In: Sharygina and Veith [104], pp. 559–575
33. Chatterjee, K., Jurdzinski, M., Henzinger, T.: Simple stochastic parity games. In: Baaz, M., Makowsky, J.A. (eds.) Proceedings of the International Conference for Computer Science Logic (CSL). LNCS, vol. 2803, pp. 100–113. Springer, Heidelberg (2003)
34. Chatterjee, K., Jurdzinski, M., Henzinger, T.: Quantitative stochastic parity games. In: Munro, J.I. (ed.) Proceedings of the Annual Symposium on Discrete Algorithms (SODA), pp. 121–130. SIAM, Philadelphia (2004)
35. Chatterjee, K., Majumdar, R., Henzinger, T.A.: Markov decision processes with multiple objectives. In: Durand, B., Thomas, W. (eds.) STACS. LNCS, vol. 3884, pp. 325–336. Springer, Heidelberg (2006)
36. Chen, T., Forejt, V., Kwiatkowska, M., Parker, D., Simaitis, A.: Automatic verification of competitive stochastic systems. Form. Methods Syst. Des. **43**(1), 61–92 (2013)
37. Chen, T., Hahn, E.M., Han, T., Kwiatkowska, M., Qu, H., Zhang, L.: Model repair for Markov decision processes. In: Proc. 7th International Symposium on Theoretical Aspects of Software Engineering (TASE'13), pp. 85–92. IEEE, Piscataway (2013)

38. Ciesinski, F., Baier, C.: LiQuor: a tool for qualitative and quantitative linear time analysis of reactive systems. In: Proc. QEST 2007, pp. 131–132. IEEE, Piscataway (2007)
39. Ciocchetta, F., Hillston, J.: Bio-PEPA: a framework for the modelling and analysis of biological systems. Theor. Comput. Sci. **410**(33–34), 3065–3084 (2009)
40. Courcoubetis, C., Yannakakis, M.: The complexity of probabilistic verification. J. ACM **42**(4), 857–907 (1995)
41. Daniele, M., Giunchiglia, F., Vardi, M.: Improved automata generation for linear temporal logic. In: Halbwachs, N., Peled, D. (eds.) Proc. International Conference on Computer Aided Verification (CAV). LNCS, vol. 1633, pp. 249–260. Springer, Heidelberg (1999)
42. Delahaye, B., Caillaud, B., Legay, A.: Probabilistic contracts: a compositional reasoning methodology for the design of stochastic systems. In: Proc. 10th Int. Conf. Application of Concurrency to System Design (ACSD'10), pp. 223–232. IEEE, Piscataway (2010)
43. Delahaye, B., Katoen, J.P., Larsen, K., Legay, A., Pedersen, M., Sher, F., Wasowski, A.: Abstract probabilistic automata. In: Jhala, R., Schmidt, D.A. (eds.) 12th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI). LNCS, vol. 6538, pp. 324–339. Springer, Heidelberg (2011)
44. Donaldson, A., Miller, A.: Symmetry reduction for probabilistic model checking using generic representatives. In: Graf, S., Zhang, W. (eds.) Proc. 4th Int. Symp. Automated Technology for Verification and Analysis (ATVA'06). LNCS, vol. 4218, pp. 9–23. Springer, Heidelberg (2006)
45. Duflot, M., Kwiatkowska, M., Norman, G., Parker, D.: A formal analysis of Bluetooth device discovery. Int. J. Softw. Tools Technol. Transf. **8**(6), 621–632 (2006)
46. Emerson, E.A.: Temporal and modal logic. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, vol. B: Formal Models and Semantics, pp. 996–1072. Elsevier, Amsterdam (1990). Chap. 14
47. Esparza, J., Křetínský, J.: From LTL to deterministic automata: a Safraless compositional approach. In: Biere, A., Bloem, R. (eds.) Computer Aided Verification—Proceedings of the 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18–22, 2014. LNCS, vol. 8559, pp. 192–208. Springer, Heidelberg (2014)
48. Etessami, K., Kwiatkowska, M.Z., Vardi, M.Y., Yannakakis, M.: Multi-objective model checking of Markov decision processes. Log. Methods Comput. Sci. **4**(4) (2008)
49. Etessami, K., Yannakakis, M.: Model checking of recursive probabilistic systems. ACM Trans. Comput. Log. **13**(2), 1–40 (2012)
50. Feller, W.: An Introduction to Probability Theory and Its Applications. Wiley, New York (1950)
51. Feng, L., Kwiatkowska, M., Parker, D.: Compositional verification of probabilistic systems using learning. In: Proc. 7th Int. Conf. Quantitative Evaluation of Systems (QEST'10), pp. 133–142. IEEE, Piscataway (2010)
52. Forejt, V., Kwiatkowska, M., Norman, G., Parker, D.: Automated verification techniques for probabilistic systems. In: Bernardo, M., Issarny, V. (eds.) Formal Methods for Eternal Networked Software Systems (SFM'11). LNCS, vol. 6659, pp. 53–113. Springer, Heidelberg (2011)
53. Forejt, V., Kwiatkowska, M., Norman, G., Parker, D., Qu, H.: Quantitative multi-objective verification for probabilistic systems. In: Abdulla, P., Leino, K. (eds.) Proc. 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'11). LNCS, vol. 6605, pp. 112–127. Springer, Heidelberg (2011)
54. Fujita, M., McGeer, P.C., Yang, J.C.Y.: Multi-terminal binary decision diagrams: an efficient data structure for matrix representation. Form. Methods Syst. Des. **10**(2/3), 149–169 (1997)
55. van Glabbeek, R., Smolka, S.A., Steffen, B., Tofts, C.M.N.: Reactive, generative, and stratified models of probabilistic processes. In: Proc. 5th Annual Symposium on Logic in Computer Science (LICS), pp. 130–141. IEEE, Piscataway (1990)

56. Grädel, E., Thomas, W., Wilke, T. (eds.): Automata, Logics, and Infinite Games: A Guide to Current Research [outcome of a Dagstuhl seminar, February 2001]. LNCS, vol. 2500. Springer, Heidelberg (2002)
57. Hahn, E., Hermanns, H., Wachter, B., Zhang, L.: PARAM: a model checker for parametric Markov models. In: Touili, T., Cook, B., Jackson, P. (eds.) 22nd International Conference on Computer Aided Verification (CAV). LNCS, vol. 6174, pp. 660–664. Springer, Heidelberg (2010)
58. Hahn, E.M., Hermanns, H., Wachter, B., Zhang, L.: PASS: abstraction refinement for infinite probabilistic models. In: Esparza, J., Majumdar, R. (eds.) Proc. TACAS 2010. LNCS, vol. 6015, pp. 353–357. Springer, Heidelberg (2010)
59. Hansson, H., Jonsson, B.: A logic for reasoning about time and reliability. Form. Asp. Comput. **6**(5), 512–535 (1994)
60. Hartonas-Garmhausen, V., Campos, S., Clarke, E.: ProbVerus: probabilistic symbolic model checking. In: Katoen, J. (ed.) 5th International AMAST Workshop on Formal Methods for Real-Time and Probabilistic Systems (ARTS). LNCS, vol. 1601, pp. 96–110. Springer, Heidelberg (1999)
61. Haverkort, B.: Performance of Computer Communication Systems: A Model-Based Approach. Wiley, Chichester (1998)
62. Heath, J., Kwiatkowska, M., Norman, G., Parker, D., Tymchyshyn, O.: Probabilistic model checking of complex biological pathways. Theor. Comput. Sci. **319**(3), 239–257 (2008)
63. Henzinger, T.A., Mateescu, M.: Propagation models for computing biochemical reaction networks. In: Fages, F. (ed.) Proc. CMSB'11, pp. 1–3. ACM, New York (2011)
64. Hermanns, H., Katoen, J.P.: The how and why of interactive Markov chains. In: de Boer, F.S., Bonsangue, M.M., Hallerstede, S., Leuschel, M. (eds.) FMCO'09. LNCS, vol. 6286, pp. 311–337. Springer, Heidelberg (2010)
65. Hermanns, H., Segala, R. (eds.): Proc. 2nd Joint Int. Workshop Process Algebra and Probabilistic Methods, Performance Modeling and Verification (PAPM-PROBMIV). LNCS, vol. 2399. Springer, Heidelberg (2002)
66. Hurd, J., McIver, A., Morgan, C.: Probabilistic guarded commands mechanized in HOL. Theor. Comput. Sci. **346**(1), 96–112 (2005)
67. Huth, M., Kwiatkowska, M.: Quantitative analysis and model checking. In: Proc. 12th Annual IEEE Symposium on Logic in Computer Science (LICS'97), pp. 111–122. IEEE, Piscataway (1997)
68. Itai, A., Rodeh, M.: Symmetry breaking in distributed networks. Inf. Comput. **88**(1), 60–87 (1990)
69. Iyer, P., Narasimha, M.: Probabilistic lossy channel systems. In: Bidoit, M., Dauchet, M. (eds.) TAPSOFT '97: Theory and Practice of Software Development. LNCS, vol. 1214, pp. 667–681. Springer, Heidelberg (1997)
70. Jeannet, B., d'Argenio, P.R., Larsen, K.G.: Rapture: A tool for verifying Markov Decision Processes. In: Tools Day'02, Technical Report. Masaryk University, Brno (2002)
71. Jonsson, B., Larsen, K., Yi, W.: Probabilistic extensions of process algebras. In: Bergstra, J.A., Pomse, A., Smolka, S.A. (eds.) Handbook of Process Algebra, pp. 685–710. Elsevier, Amsterdam (2001)
72. Karloff, H.: Linear Programming. Birkhäuser, Boston (1991)
73. Kattenbelt, M., Kwiatkowska, M., Norman, G., Parker, D.: Abstraction refinement for probabilistic software. In: Jones, N., Müller-Olm, M. (eds.) Proc. 10th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'09). LNCS, vol. 5403, pp. 182–197. Springer, Heidelberg (2009)
74. Kattenbelt, M., Kwiatkowska, M., Norman, G., Parker, D.: A game-based abstraction-refinement framework for Markov decision processes. Form. Methods Syst. Des. **36**(3), 246–280 (2010)
75. Kemeny, J., Snell, J.: Finite Markov Chains. Van Nostrand, Princeton (1960)

76. Komárková, Z., Křetínský, J.: Rabinizer 3: Safraless translation of LTL to small deterministic automata. In: Cassez, F., Raskin, J. (eds.) Automated Technology for Verification and Analysis—Proceedings of the 12th International Symposium, ATVA 2014, Sydney, NSW, Australia, November 3–7, 2014. LNCS, vol. 8837, pp. 235–241. Springer, Heidelberg (2014)
77. Kulkarni, V.: Modeling and Analysis of Stochastic Systems. Chapman & Hall, London (1995)
78. Kwiatkowska, M., Norman, G., Parker, D.: Symmetry reduction for probabilistic model checking. In: Ball, T., Jones, R.B. (eds.) Proc. of the 18th International Conference on Computer Aided Verification (CAV). LNCS, vol. 4144, pp. 234–248. Springer, Heidelberg (2006)
79. Kwiatkowska, M., Norman, G., Parker, D.: Using probabilistic model checking in systems biology. ACM SIGMETRICS Perform. Eval. Rev. **35**(4), 14–21 (2008)
80. Kwiatkowska, M., Norman, G., Parker, D.: Stochastic games for verification of probabilistic timed automata. In: Ouaknine, J., Vaandrager, F.W. (eds.) Proc. 7th International Conference on Formal Modelling and Analysis of Timed Systems (FORMATS'09). LNCS, vol. 5813, pp. 212–227. Springer, Heidelberg (2009)
81. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) Proc. 23rd International Conference on Computer Aided Verification (CAV'11). LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011)
82. Kwiatkowska, M., Norman, G., Parker, D., Qu, H.: Assume-guarantee verification for probabilistic systems. In: Esparza, R.M.J. (ed.) 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 6015, pp. 23–37 (2010)
83. Kwiatkowska, M., Norman, G., Parker, D., Qu, H.: Compositional probabilistic verification through multi-objective model checking. Inf. Comput. **232**, 38–65 (2013)
84. Kwiatkowska, M., Norman, G., Segala, R., Sproston, J.: Automatic verification of real-time systems with discrete probability distributions. Theor. Comput. Sci. **282**, 101–150 (2002)
85. Kwiatkowska, M., Norman, G., Sproston, J.: Probabilistic model checking of the IEEE 802.11 wireless local area network protocol. In: Hermanns, H., Segala, R. (eds.) Proc. 2nd Joint International Workshop on Process Algebra and Probabilistic Methods, Performance Modeling and Verification (PAPM/PROBMIV'02). LNCS, vol. 2399, pp. 169–187. Springer, Heidelberg (2002)
86. Kwiatkowska, M., Parker, D., Qu, H.: Incremental quantitative verification for Markov decision processes. In: Proc. IEEE/IFIP International Conference on Dependable Systems and Networks (DSN-PDS'11), pp. 359–370. IEEE, Piscataway (2011)
87. Larsen, K., Skou, A.: Bisimulation through probabilistic testing. Inf. Comput. **94**(1), 1–28 (1991)
88. Lassaigne, R., Peyronnet, S.: Approximate verification of probabilistic systems. In: [65], pp. 213–214 (2002)
89. Lassaigne, R., Peyronnet, S.: Approximate planning and verification for large Markov decision processes. In: Proceedings of the 27th Annual ACM Symposium on Applied Computing, SAC '12, pp. 1314–1319. ACM, New York (2012)
90. Lynch, N.: Distributed Algorithms. Morgan Kaufmann, San Francisco (1996)
91. McIver, A., Morgan, C.: Games, probability and the quantitative μ-calculus qMμ. In: Baaz, M., Voronkov, A. (eds.) Proc. LPAR 2002. LNCS, vol. 2514, pp. 292–310. Springer, Heidelberg (2002)
92. Mikeev, L., Sandmann, W., Wolf, V.: Numerical approximation of rare event probabilities in biochemically reacting systems. In: Gupta, A., Henzinger, T.A. (eds.) Proc. CMSB 2013. LNCS, vol. 8130, pp. 5–18. Springer, Heidelberg (2013)
93. Mio, M.: Probabilistic modal μ-calculus with independent product. Log. Methods Comput. Sci. **8**(4), 1–36 (2012)

94. Norman, G., Parker, D., Kwiatkowska, M., Shukla, S., Gupta, R.: Using probabilistic model checking for dynamic power management. In: Leuschel, M., Gruner, S., Presti, S.L. (eds.) Proc. 3rd Workshop on Automated Verification of Critical Systems (AVoCS'03), Technical Report DSSE-TR-2003-2, University of Southampton, pp. 202–215 (2003)
95. Norman, G., Parker, D., Sproston, J.: Model checking for probabilistic timed automata. Form. Methods Syst. Des. **43**(2), 164–190 (2013)
96. Norris, J.R.: Markov chains. Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press, Cambridge (1998)
97. Pnueli, A., Zuck, L.: Probabilistic verification by tableaux. In: Proc. Annual Symposium on Logic in Computer Science (LICS), pp. 322–331. IEEE, Piscataway (1986)
98. PRISM web site. www.prismmodelchecker.org. Accessed 20 August 2013
99. Puterman, M.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. Wiley, New York (1994)
100. Schrijver, A.: Combinatorial Optimization: Polyhedra and Efficiency. Springer, Heidelberg (2003)
101. Segala, R.: Modeling and verification of randomized distributed real-time systems. Ph.D. thesis, Massachusetts Institute of Technology (1995)
102. Segala, R., Lynch, N.: Probabilistic simulations for probabilistic processes. In: Jonsson, B., Parrow, J. (eds.) Proc. CONCUR '94. LNCS, vol. 836, pp. 481–496. Springer, Heidelberg (1994)
103. Sen, K., Viswanathan, M., Agha, G.: Model-checking Markov chains in the presence of uncertainties. In: Hermanns, H., Palsberg, J. (eds.) 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 3920, pp. 394–410. Springer, Heidelberg (2006)
104. Sharygina, N., Veith, H. (eds.): Computer Aided Verification—Proceedings of the 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13–19, 2013. LNCS, vol. 8044. Springer, Heidelberg (2013)
105. Vardi, M.Y.: Automatic verification of probabilistic concurrent finite-state programs. In: Proc. 26th IEEE Symposium on Foundations of Computer Science (FOCS), pp. 327–338. IEEE, Piscataway (1985)
106. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification. In: Symposium on Logic in Computer Science (LICS'86), pp. 332–345. IEEE, Piscataway (1986)
107. Vardi, M.Y., Wolper, P.: Reasoning about infinite computations. Inf. Comput. **115**(1), 1–37 (1994)
108. Younes, H., Kwiatkowska, M., Norman, G., Parker, D.: Numerical vs. statistical probabilistic model checking. Int. J. Softw. Tools Technol. Transf. **8**(3), 216–228 (2006)
109. Younes, H., Simmons, R.: Probabilistic verification of discrete event systems using acceptance sampling. In: Brinksma, E., Larsen, K.G. (eds.) Proc. 14th International Conference on Computer Aided Verification (CAV). LNCS, vol. 2404, pp. 223–235. Springer, Heidelberg (2002)
110. Zhang, L., Hermanns, H.: Deciding simulations on probabilistic automata. In: Namjoshi, K., Yoneda, T., Higashino, T., Okamura, Y. (eds.) Proc. 5th Int. Symp. Automated Technology for Verification and Analysis (ATVA'07). LNCS, vol. 4762, pp. 207–222. Springer, Heidelberg (2007)

Chapter 29
Model Checking Real-Time Systems

Patricia Bouyer, Uli Fahrenberg, Kim Guldstrand Larsen, Nicolas Markey, Joël Ouaknine, and James Worrell

Abstract This chapter surveys timed automata as a formalism for model checking real-time systems. We begin with introducing the model, as an extension of finite-state automata with real-valued variables for measuring time. We then present the main model-checking results in this framework, and give a hint about some recent extensions (namely weighted timed automata and timed games).

29.1 Introduction

Timed automata were introduced by Rajeev Alur and David Dill in the early 1990s [13] as finite-state automata equipped with real-valued variables for measuring time between transitions in the automaton. These variables all evolve at the same rate; they can be reset along some transitions, and used as *guards* along other transitions or invariants to be preserved while letting time elapse in locations of the automaton.

Timed automata have proven very convenient for modeling and reasoning about real-time systems: they combine a powerful formalism with advanced expressiveness and efficient algorithmic and tool support, and have become a model of choice

P. Bouyer · N. Markey (✉)
LSV, CNRS & ENS Paris-Saclay, Cachan, France
e-mail: nicolas.markey@irisa.fr

N. Markey
IRISA, CNRS & INRIA & Univ. Rennes 1, Rennes, France

U. Fahrenberg
École Polytechnique, Palaiseau, France

K.G. Larsen
Aalborg University, Aalborg, Denmark

J. Ouaknine · J. Worrell
University of Oxford, Oxford, UK

J. Ouaknine
Max Planck Institute for Software Systems, Saarbrücken, Germany

E.M. Clarke et al. (eds.), *Handbook of Model Checking*,
DOI 10.1007/978-3-319-10575-8_29

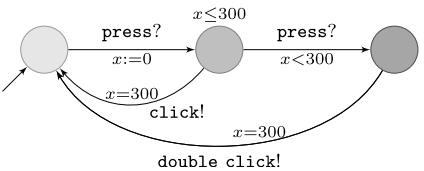


Fig. 1 A timed automaton

in the framework of verification of embedded systems. The timed-automata formalism is now routinely applied to the analysis of real-time control programs [85, 127] and timing analysis of software and asynchronous circuits [60, 146, 150]. Similarly, numerous real-time communication protocols have been analysed using timed automata technology, often with inconsistencies being revealed [95, 144]. During the last few years, timed-automata-based schedulability and response-time analysis of multitasking applications running under real-time operating systems have received substantial research effort [61, 78, 80, 109, 156]. Also, for optimal planning and scheduling, (priced) timed automata technology has been shown to provide competitive and complementary performances with respect to classical approaches [1, 2, 32, 86, 97, 108, 119]. Finally, controller synthesis from timed games has been applied to a number of industrial case studies [6, 71, 110].

The handiness of this formalism is exemplified in Fig. 1, modeling a (simplified) computer mouse: this automaton receives `press` events, corresponding to an action of the user on the button of the mouse. When two such events are close enough (less than 300 milliseconds apart), this is translated into a `double_click` event.

Because clock variables are real-valued, timed automata are in fact infinite-state models, where a configuration is given by a location of the automaton and a valuation of the clocks. Timed automata have two kinds of transitions: *action transitions* correspond to firing a transition of the automaton, and *delay transitions* correspond to letting time elapse in the current location of the automaton. Section 29.2 provides the definitions of this framework. The main technical ingredient for dealing with this infinity of states is the *region abstraction*, as we explain in Sect. 29.3. Roughly, two clock valuations are called *region equivalent* whenever they satisfy the exact same set of constraints of the form $x - y \bowtie c$, where the difference of two clocks x and y is compared to some integer c (no greater than some constant M). This abstraction can be used to develop various algorithms, in particular for deciding bisimilarity (Sect. 29.4) or model checking some quantitative extensions of the classical temporal logics CTL and LTL (Sect. 29.6). We also show that some problems are undecidable, most notably language containment (Sect. 29.5) and model checking the full quantitative extension of LTL. On the practical side, regions are in some sense too fine-grained, and another abstraction, called *zones*, is preferred for implementation purposes. Roughly, zones provide a way of grouping many regions together, which is often relevant in practical situations. We explain in Sect. 29.7 how properties of timed automata can be verified in practice.

Finally, we conclude this chapter with two powerful extensions of timed automata: first, *weighted timed automata* allow for modeling quantitative constraints beyond time; since resource (e.g., energy) consumption is usually tightly bound to time elapsing, timed automata provide a convenient framework for modeling such

quantitative aspects of systems. Unlike hybrid systems (see Chap. 30), weighted timed automata still enjoy some nice decidability properties (in restricted settings though), as we explain in Sect. 29.8. Then in Sect. 29.9 we present *timed games*, which are very powerful and convenient for dealing with the controller synthesis problem (see Chap. 27) in a timed framework. Timed games also provide an interesting way of modeling uncertainty in real-time systems, assuming worst-case resolution of the uncertainty while still trying to benefit from non-worst-case situations.

29.2 Timed Automata

In this chapter, we consider as time domain the set $\mathbb{R}_{\geq 0}$ of non-negative reals. While discrete time might look reasonable for representing digital systems, it assumes synchronous interactions between the systems. We refer to [17, 27, 67, 102] for more discussions on this point.

Let Σ be a finite set of *actions*. A *time sequence* is a finite or infinite non-decreasing sequence of non-negative reals. A *timed word* is a finite or infinite sequence of pairs $(a_1, t_1) \dots (a_p, t_p) \dots$ such that $a_i \in \Sigma$ for every i, and $(t_i)_{i \geq 1}$ is a time sequence. An infinite timed word is *converging* if its time sequence is bounded above (or, equivalently, converges).

We consider a finite set C of variables, called *clocks*. A *(clock) valuation* over C is a mapping $v \colon C \to \mathbb{R}_{\geq 0}$ which assigns to each clock a real value. The set of all clock valuations over C is denoted $\mathbb{R}_{\geq 0}^C$, and $\mathbf{0}_C$ denotes the valuation assigning 0 to every clock $x \in C$.

Let $v \in \mathbb{R}_{\geq 0}^C$ be a valuation and $t \in \mathbb{R}_{\geq 0}$; the valuation $v + t$ is defined by $(v + t)(x) = v(x) + t$ for every $x \in C$. For a subset r of C, we denote by $v[r]$ the valuation obtained from v by resetting clocks in r; formally, for every $x \in r$, $v[r](x) = 0$ and for every $x \in C \setminus r$, $v[r](x) = v(x)$.

The set $\Phi(C)$ of *clock constraints* over C is defined by the grammar

$$\Phi(C) \ni \varphi ::= x \bowtie k \mid \varphi_1 \wedge \varphi_2 \quad \big(x \in C,\ k \in \mathbb{Z} \text{ and } \bowtie \in \{<, \leq, =, \geq, >\}\big).$$

We will sometimes make use of *diagonal clock constraints*, which additionally allow constraints of the form $x - y \bowtie k$. We write $\Phi_d(C)$ for the extension of $\Phi(C)$ with diagonal constraints. If $v \in \mathbb{R}_{\geq 0}^C$ is a clock valuation, we write $v \models \varphi$ when v satisfies the clock constraint φ, and we say that v satisfies $x \bowtie k$ whenever $v(x) \bowtie k$ (similarly, v satisfies $x - y \bowtie k$ when $v(x) - v(y) \bowtie k$). If φ is a clock constraint, we write $[\![\varphi]\!]_C$ for the set of clock valuations $\{v \in \mathbb{R}_{\geq 0}^C \mid v \models \varphi\}$.

Definition 1 ([13]) A *timed automaton* is a tuple $(L, \ell_0, C, \Sigma, I, E)$ consisting of a finite set L of locations with initial location $\ell_0 \in L$, a finite set C of clocks, an invariant[1] mapping $I \colon L \to \Phi(C)$, a finite alphabet Σ and a set $E \subseteq L \times \Phi(C) \times \Sigma \times 2^C \times L$ of edges. We shall write $\ell \xrightarrow{\varphi, a, r} \ell'$ for an edge $(\ell, \varphi, a, r, \ell') \in E$;

[1] The original definition of timed automata [13] did not contain invariants in locations, but had Büchi conditions to enforce liveness. Invariants were added by [103]. Several other convenient extensions have been introduced since then, which we discuss in Sect. 29.3.2.

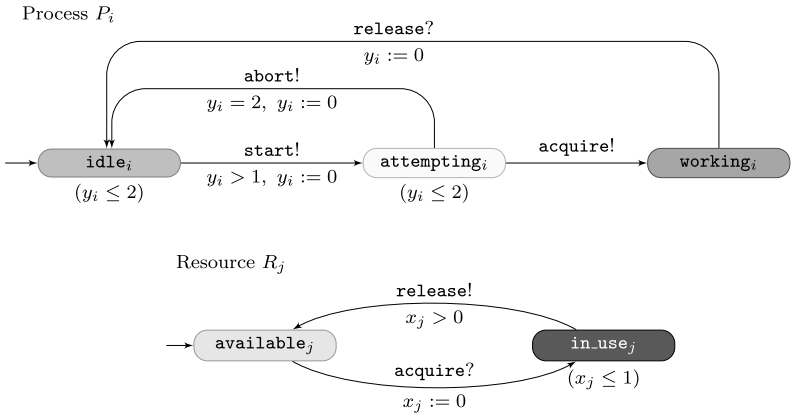


Fig. 2 Model of a process that acquires and releases two resources. Here and in the rest of this chapter, transitions are decorated with their associated guards (e.g., $x_j > 0$), letters of the alphabet (e.g., `release!`), and resets (written e.g., as $x_j := 0$); invariants (if any) are written in brackets below their corresponding locations

formula φ is the *guard* of the transition (and has to be satisfied when the transition is taken), and r is the set of clocks that are set to zero after taking that transition.

Later for defining languages accepted by timed automata we may add final or repeated (Büchi) locations, and for defining logical satisfaction relations we may add atomic proposition labeling to timed automata. However for readability reasons we omit them here.

Example 1 Figure 2 shows timed automata models for processes and resources. Processes can use resources, but mutual exclusion is expected. The model for process P_i is given in the upper part of the figure, whereas the model for resource R_j is given in the lower part of the figure. Starting in the `idle` location, the process should start within one to two time units requesting a resource. After two time units it must abort its request, unless before two time units it acquires the resource and goes to the `working` location. The resource is released when the process is done working with it.

Our model for a resource has two locations, and when the resource is available, it can be acquired and should be released within one time unit.

The *operational semantics* of a timed automaton $A = (L, \ell_0, C, \Sigma, I, E)$ is the (infinite-state) timed transition system $[\![A]\!] = (S, s_0, \mathbb{R}_{\geq 0} \times \Sigma, T)$ given as follows:

$$S = \left\{(\ell, v) \in L \times \mathbb{R}_{\geq 0}^{C} \;\middle|\; v \models I(\ell)\right\} \qquad s_0 = (\ell_0, \mathbf{0}_C)$$

$$T = \Big\{(\ell, v) \xrightarrow{d,a} \left(\ell', v'\right) \;\Big|\; \forall d' \in [0, d] : v + d' \models I(\ell),$$
$$\text{and } \exists \ell \xrightarrow{\varphi,a,r} \ell' \in E :\ v + d \models \varphi,\ \text{and } v' = (v + d)[r]\Big\}$$

In words, one can jump from one state (ℓ, v) to another one (ℓ', v') by selecting a delay to be elapsed in ℓ (provided that the invariant of location ℓ is fulfilled in the meantime) and an edge of the automaton, which is taken after the delay, provided that its guard is satisfied at that time. In this semantics, a transition combines both a delay and (followed by) the application of an edge of the automaton. A slightly different semantics is sometimes used, which distinguishes pure-delay transitions (denoted $\xrightarrow{d}$, for $d \in \mathbb{R}_{\geq 0}$) and pure-action transitions (denoted $\xrightarrow{a}$ with $a \in \Sigma$).

A (finite or infinite) *run* of a timed automaton A is a (finite or infinite) path $\rho = (\ell_0, v_0) \xrightarrow{d_1,a_1} (\ell_1, v_1) \xrightarrow{d_2,a_2} \cdots$ in the transition system $[\![A]\!]$, which starts with $v_0 = \mathbf{0}_C$. Given a run $\rho = (\ell_0, v_0) \xrightarrow{d_1,a_1} (\ell_1, v_1) \xrightarrow{d_2,a_2} (\ell_2, v_2) \ldots$, we say that it reads the timed word $w = (a_1, t_1)(a_2, t_2) \ldots$ where for every i, $t_i = \sum_{j \leq i} d_j$. A run is *time-divergent* if its time sequence $(t_i)_i$ diverges. A timed automaton is *non-Zeno* if any finite run can be extended into a time-divergent run.

Example 2 The process R_1 given in Fig. 2 has a single clock x_1, and has as set of states $S = \{\texttt{available}_1\} \times \mathbb{R}_{\geq 0} \cup \{\texttt{in\_use}_1\} \times [0, 1]$ where we identify valuations (for the single clock x_1) with the value of x_1. We give below a possible run for the resource R_1:

$$(\texttt{available}_1, 0) \xrightarrow{5.4,\texttt{acquire?}} (\texttt{in\_use}_1, 0) \xrightarrow{0.8,\texttt{release!}} (\texttt{available}_1, 0.8) \xrightarrow{1.4,\texttt{acquire?}} (\texttt{in\_use}_1, 0) \to \cdots$$

In location `in_use`, the invariant is satisfied in this run because the value of x_1 never exceeds 0.8 (hence satisfies the constraint $x_1 \leq 1$).

We now define the parallel composition of timed automata, which allows us to define systems in a compositional way [23, 107]. Let $(A_i)_{1 \leq i \leq n}$ be n timed automata, where $A_i = (L_i, \ell_0^i, C_i, \Sigma_i, I_i, E_i)$. Assume that all Σ_i's are disjoint, and all C_i's are disjoint. If Σ is a new alphabet, given a (partial) synchronization function $f : \prod_{i=1}^n (\Sigma_i \cup \{-\}) \to \Sigma$, the *synchronized product* (or *parallel composition*) $(A_1 \parallel A_2 \parallel \cdots \parallel A_n)_f$ is the timed automaton $A = (L, \ell_0, C, \Sigma, I, E)$ where $L = L_1 \times \cdots \times L_n$, $\ell_0 = (\ell_0^1, \ldots, \ell_0^n)$, $C = C_1 \cup \cdots \cup C_n$, $I((\ell_1, \ldots, \ell_n)) = \bigwedge_{i=1}^n I_i(\ell_i)$ for every $(\ell_1, \ldots, \ell_n) \in L_1 \times \cdots \times L_n$, and the set E is composed of the transitions $(\ell_1, \ldots, \ell_n) \xrightarrow{\varphi,a,r} (\ell'_1, \ldots, \ell'_n)$ whenever

1. there exists $(\alpha_1, \ldots, \alpha_n) \in \prod_{i=1}^n (\Sigma_i \cup \{-\})$ such that $f(\alpha_1, \ldots, \alpha_n) = a$;
2. if $\alpha_i = -$, then $\ell'_i = \ell_i$;
3. if $\alpha_i \neq -$, then there is a transition $\ell_i \xrightarrow{\varphi_i,\alpha_i,r_i} \ell'_i$ in E_i
4. $\varphi = \bigwedge \{\varphi_i \mid \alpha_i \neq -\}$ and $r = \bigcup \{r_i \mid \alpha_i \neq -\}$

Example 3 We build on the system given in Fig. 2. The process and the resources are not expected to run independently, but they are part of a global system where the process should synchronize with the resources. Hence for this system we have a natural synchronization function f defined by Table 1.

Table 1 The synchronization function f

P_1	P_2	R_1		
start!	–	–	→	start_1
–	start!	–	→	start_2
abort!	–	–	→	abort_1
–	abort!	–	→	abort_2
acquire!	–	acquire?	→	acquire_1
–	acquire!	acquire?	→	acquire_2
release?	–	release!	→	release_1
–	release?	release!	→	release_2

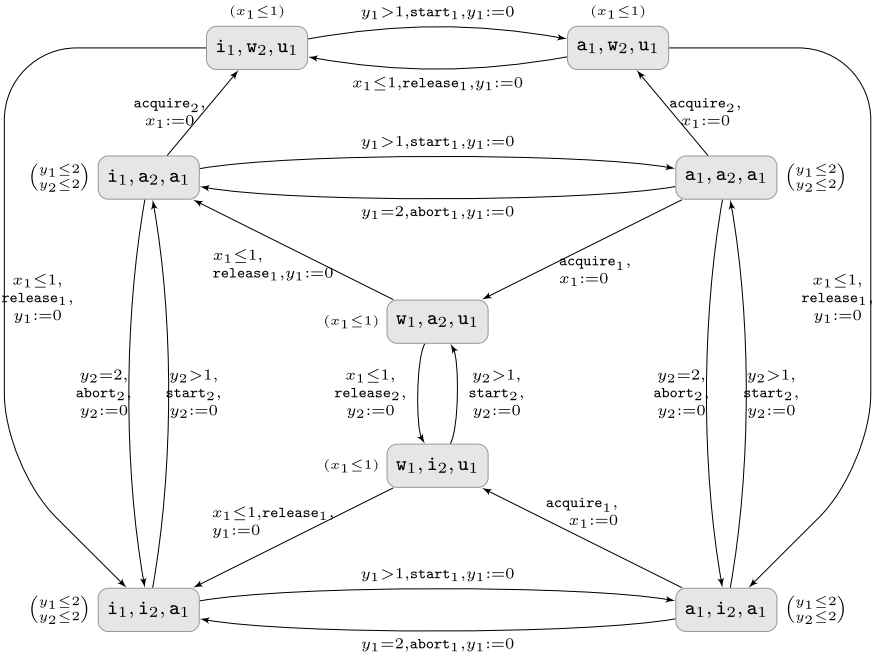


Fig. 3 The global system $(P_1 \parallel P_2 \parallel R_1)_f$, where "$\text{i}_j$" (resp. "$\text{a}_j$", "$\text{w}_j$") stands for location "$\text{idle}_j$" (resp. "$\text{attempting}_j$", "$\text{working}_j$") in P_j, and "a_1" (resp. "u_1") stands for location "available_1" (resp. "in_use_1") in R_1

The global system $(P_1 \parallel P_2 \parallel R_1)_f$ (more precisely the part which is reachable from the initial state) is depicted in Fig. 3. This automaton is rather complex, and the component-based definition as $(P_1 \parallel P_2 \parallel R_1)_f$ is much easier to understand. Furthermore this allows addition of other processes and other resources to the system without any effort.

29.3 Checking Reachability

In this section we are interested in the most basic problem regarding timed automata, namely reachability. This problem asks, given a timed automaton A, whether a distinguished set of locations F of A is reachable or not.

29.3.1 Region Equivalence

For the rest of this section we fix a timed automaton $A = (L, \ell_0, C, \Sigma, I, E)$ and a set of target locations F. For every clock $x \in C$ we let M_x be the maximal constant clock x is compared to in A.

Two valuations $v, v' \colon C \to \mathbb{R}_{\geq 0}$ are said to be *region equivalent w.r.t. maximal constants* $M = (M_x)_{x \in C}$, denoted $v \cong_M v'$, if [2]

- for all $x \in C$, $\lfloor v(x) \rfloor = \lfloor v'(x) \rfloor$ or $v(x), v'(x) > M_x$, and
- for all $x \in C$ with $v(x) \leq M_x$, $\langle v(x) \rangle = 0$ iff $\langle v'(x) \rangle = 0$, and
- for all $x, y \in C$ with $v(x) \leq M_x$ and $v(y) \leq M_y$, $\langle v(x) \rangle \leq \langle v(y) \rangle$ iff $\langle v'(x) \rangle \leq \langle v'(y) \rangle$.

The equivalence classes of valuations with respect to $\cong_M$ are called *regions* (with maximal constants M). The number of regions is finite and is bounded above by $n! \cdot 2^n \cdot \prod_{x \in C}(2M_x + 2)$. Region equivalence of valuations is extended to states of A by declaring that $(\ell, v) \cong_M (\ell', v')$ whenever $\ell = \ell'$ and $v \cong_M v'$. We write $[\ell, v]_{\cong_M}$ for the equivalence class of (ℓ, v).

Region equivalence enjoys nice properties, the most important of which is that it is a *time-abstracted bisimulation* in the following sense:

Definition 2 A relation R on the states of A is a *time-abstracted bisimulation* if $(\ell_1, v_1) R (\ell_2, v_2)$ and $(\ell_1, v_1) \xrightarrow{d_1, a} (\ell_1', v_1')$ for some $d_1 \in \mathbb{R}_{\geq 0}$ and $a \in \Sigma$ imply $(\ell_2, v_2) \xrightarrow{d_2, a} (\ell_2', v_2')$ for some $d_2 \in \mathbb{R}_{\geq 0}$, with $(\ell_1', v_1') R (\ell_2', v_2')$ and vice versa.

In other words, from two equivalent states, the automaton can take the same transitions, except that the values of the delays might have to be changed. This fundamental property has important consequences, like the construction of an interesting finite abstraction for A.

Definition 3 The *region automaton* $\mathcal{R}_{\cong_M}(A) = (S, s_0, \Sigma, T)$ associated with A has as set of states the quotient $S = (L \times \mathbb{R}_{\geq 0}^C)/_{\cong_M}$, as initial state $s_0 = [\ell_0, \mathbf{0}_C]_{\cong_M}$, and as transitions all the $[\ell, v]_{\cong_M} \xrightarrow{a} [\ell', v']_{\cong_M}$ for which $(\ell, v) \xrightarrow{d, a} (\ell', v')$ for some $d \in \mathbb{R}_{\geq 0}$. The target set of $\mathcal{R}_{\cong_M}(A)$ is defined as $S_F = \{[\ell, v]_{\cong_M} \mid \ell \in F\}$.

[2] For $d \in \mathbb{R}_{\geq 0}$ we write $\lfloor d \rfloor$ and $\langle d \rangle$ for the integral and fractional parts of d, i.e., $d = \lfloor d \rfloor + \langle d \rangle$.

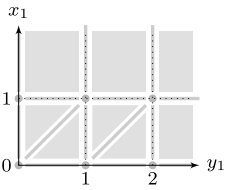


Fig. 4 Clock regions for the system $(P_1 \parallel R_1)_{f_1}$

The region automaton $\mathcal{R}_{\cong_M}(A)$ is a finite automaton whose size is exponential compared with the size of A. It can be used to check, e.g., reachability properties (or equivalently language emptiness):

Proposition 1 *The set of locations F is reachable in A from ℓ_0 iff S_F is reachable in $\mathcal{R}_{\cong_M}(A)$ from s_0.*

The region automaton has exponentially larger size, but checking a reachability property can be done on the fly, hence this can be done in polynomial space. One of the most fundamental theorems in the model checking of timed automata can be stated as follows.

Theorem 1 ([13]) *The reachability problem in timed automata is* PSPACE*-complete.*

Example 4 Restricting our running example to process P_1 and resource R_1 (we assume f_1 is the synchronization function f restricted to those two processes), the global system has two clocks, x_1 and y_1. The set of regions is then depicted in Fig. 4. There are 28 regions. The (reachable part of the) corresponding region automaton is depicted in Fig. 5. In this drawing we omit indices over names of locations since they should all be 1; also, the thick "release" transition at the top corresponds to a set of transitions from all the states on the right to all the states on the left.

29.3.2 Some Extensions of Timed Automata

Timed automata are the most basic model for representing systems with (quantitative) real-time constraints. It is natural to extend the model with features that help in modeling real systems. However decidability of the reachability problem remains the fundamental property one wants to preserve. In this subsection we mention several variants and extensions of timed automata that have been proposed in the literature.

Timed automata as defined in this chapter are the so-called *diagonal-free* timed automata since only constraints of the form $x \bowtie k$ are used. Timed automata with diagonal constraints (of the form $x - y \bowtie k$) were also originally defined in the seminal paper [13]. They can be analyzed using a slight refinement of the region automaton, but with no extra complexity. Furthermore, diagonal constraints can be

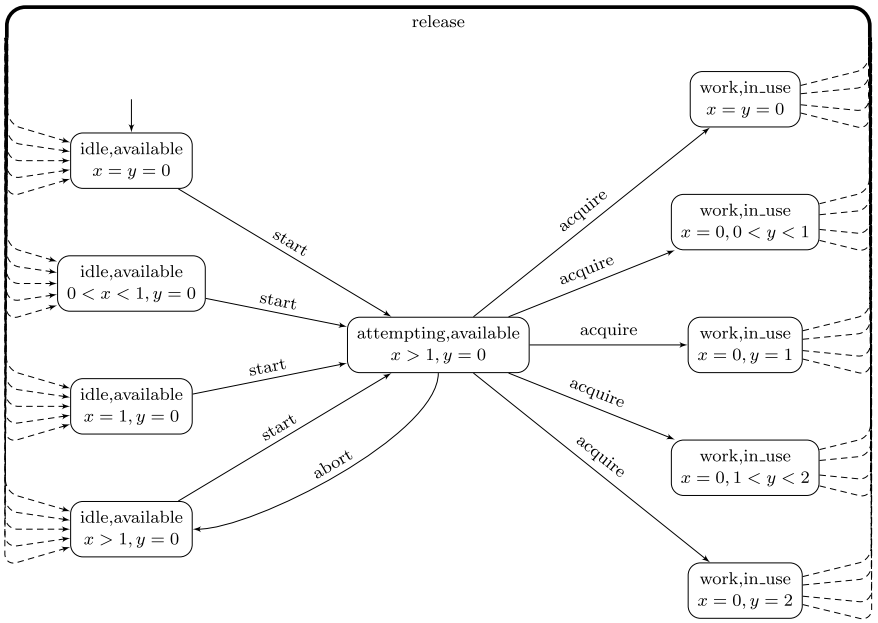


Fig. 5 Region automaton for $(P_1 \parallel R_1)_{f_1}$

removed from the model, at the expense of a (possibly exponential) blowup in the number of locations of the model [40].

Another useful extension of timed automata is obtained by allowing edges to set clocks to arbitrary positive integers ($x := k$) instead of only 0, or even to synchronize clock values ($x := y$). In [53] it is shown that any such *updatable* timed automaton can be converted to a usual one, hence this class is no more expressive than timed automata. If one also considers other updates however, like $x := x + 1$ or $x :> k$ (which non-deterministically sets x to some value larger than k), the situation is much more complex [53] and decidability of reachability is no longer preserved.

One can also extend the timed-automata formalism by allowing richer clock constraints, such as, e.g., $x + y \leq 5$ or $2x - 3y > 1$. Most such attempted extensions however lead to undecidability of the reachability problem, see for instance [41].

One can extend timed automata with *urgency* requirements [46]. For instance, some locations might be labelled as urgent, which indicates that no time can be spent in this location, it has to be left immediately when entered: an urgent location ℓ can easily be converted into a usual one by introducing an extra clock x which is reset in any edge to ℓ and has invariant $x = 0$ in ℓ, hence location-urgency does not add expressiveness to the class of timed automata. Some synchronization could be also labelled as urgent: in that case, the corresponding action should be done as soon as it is enabled. In our modeling of Example 1 the synchronization "acquire!/acquire?" could be made urgent since it is natural that a process acquires the resource as soon as it is available.

Another extension of timed automata we should mention is the *stopwatch* automata of [100]. Here timed automata are extended by allowing clocks to be stopped during a delay. Even though reachability is also undecidable for this extension and it has been shown to have the same expressive power as hybrid automata [72], stopwatch automata have found some applications, e.g., in scheduling [3, 141] and permit efficient over-approximate analysis [72]. Further extensions of the dynamics of timed automata lead to *rectangular* automata [100] and eventually to general *hybrid* automata [12, 98].

Finally, another interesting direction in which timed automata have been extended consists in adding parameters. Parameters can be used in lieu of numerical constants in the timed automaton, with the aim of deciding the existence of (and computing) values for the parameters for which a given property holds true. The use of parameters simplifies the modeling phase, but unfortunately the existence of valid parameters turns out to be undecidable in general [19]. Several decidable classes have been identified, including one-clock parametric timed automata [19, 129] and L/U automata, where each parameter can be used either in lower-bound constraints or in upper-bound constraints [106].

29.4 (Bi)simulation Checking

29.4.1 (Bi)simulations for Timed Automata

As detailed in Sect. 29.2, the operational semantics of timed automata is given in terms of timed transition systems, which in fact can be viewed as standard labelled transition systems, with labels (d, a) comprising a delay and a letter. Hence any behavioral equivalence and preorder defined on labelled transition systems may be interpreted over timed automata. In particular the classical notions of simulation and bisimulation [130, 138] give rise to the following notion of timed (bi)simulation:

Definition 4 Let $A = (L, \ell_0, C, \Sigma, I, E)$ be a timed automaton. A relation $R \subseteq L \times \mathbb{R}_{\geq 0}^C \times L \times \mathbb{R}_{\geq 0}^C$ is a *timed simulation* provided that for all $(\ell_1, v_1)\ R\ (\ell_2, v_2)$, for all $(\ell_1, v_1) \xrightarrow{d,a} (\ell_1', v_1')$ with $d \in \mathbb{R}_{\geq 0}$ and $a \in \Sigma$, there exists some (ℓ_2', v_2') such that $(\ell_1', v_1')\ R\ (\ell_2', v_2')$ and $(\ell_2, v_2) \xrightarrow{d,a} (\ell_2', v_2')$.

A *timed bisimulation* is a timed simulation which is also symmetric, and two states $(\ell_1, v_1), (\ell_2, v_2) \in [\![A]\!]$ are said to be *timed bisimilar*, written $(\ell_1, v_1) \sim (\ell_2, v_2)$, if there exists a timed bisimulation R for which $(\ell_1, v_1)\ R\ (\ell_2, v_2)$.

Note that $\sim$ is itself a timed bisimulation on A (indeed the greatest such), which is easily shown to be an equivalence relation and hence transitive, reflexive, and symmetric. Also—as usual—timed bisimilarity may be lifted to an equivalence between two timed automata A and B by relating their initial states.

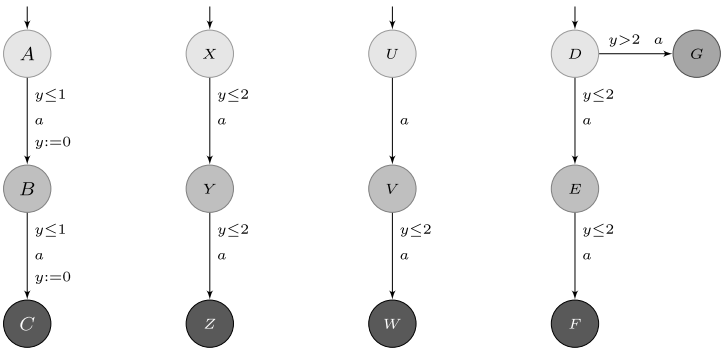


Fig. 6 Four timed automata A, X, U and D

Consider the four automata A, X, U and D in Fig. 6 (identifying the automata with the names of their initial locations). Here (U, v) and (D, v) are timed bisimilar as any transition $(U, v) \xrightarrow{d,a} (V, v')$ may be matched by either $(D, v) \xrightarrow{a} (G, v')$ or $(D, v) \xrightarrow{a} (E, v')$ depending on whether $v(y) > 2$ or not after delay d. In fact, it may easily be seen that U and D are the only locations of Fig. 6 that are timed bisimilar (when coupled with the same valuation of y). E.g., A and X are not timed bisimilar since the transition $(X, 0) \xrightarrow{1.5,a} (Y, 1.5)$ cannot be matched by $(A, 0)$ by a transition with *exactly* the same duration. Instead A and X are related by the weaker notion of *time-abstracted* bisimulation, which does not require equality of the delays (see Definition 2). It may be seen that A and X are both time-abstracted simulated by U and D but *not* time-abstracted bisimilar to U and D. Also, U and D are time-abstracted bisimilar, which follows from the following easy fact:

Theorem 2 *Any two automata being timed bisimilar are also time-abstracted bisimilar.*

29.4.2 Checking (Bi)simulations

As we now explain, timed and time-abstracted (bi)similarity are decidable for timed automata.

Theorem 3 *Time-abstracted similarity and bisimilarity are decidable for timed automata.*

For proving this result, one only needs to see that time-abstracted (bi)simulation in the timed automaton is the same as ordinary (bi)simulation in the associated region automaton; indeed, any state in $[\![A]\!]$ is untimed bisimilar to its image in $[\![A]\!]_{\cong}$. The result follows by finiteness of the region automaton.

For timed bisimilarity, decidability—as we shall see in Sect. 29.9—is obtained by playing a game on a product construction, yielding an exponential-time algorithm for checking timed bisimilarity.

Theorem 4 ([73]) *Timed similarity and bisimilarity are decidable for timed automata.*

29.5 Language-Theoretic Properties

29.5.1 Language of a Timed Automaton

This section introduces the notion of (timed) *language* associated with timed automata, and focusses on basic decision problems such as language emptiness and inclusion, as well as standard Boolean operations on languages.

Properties of languages associated with various computational models are a classical object of study in computer science; moreover, many model-checking, refinement, and verification problems can often be stated in terms of languages, notably by translating them into language emptiness or language inclusion problems.

In this section we consider timed automata augmented with sets of *accepting locations*. Given a timed automaton $A = (L, \ell_0, C, \Sigma, I, E, F)$, where $F \subseteq L$ is the set of accepting locations, a finite run

$$\rho = (\ell_0, v_0) \xrightarrow{d_1,a_1} (\ell_1, v_1) \xrightarrow{d_2,a_2} \dots \xrightarrow{d_n,a_n} (\ell_n, v_n)$$

of A is *accepting* if $\ell_n \in F$. The language $\mathcal{L}(A)$ of A consists of all finite timed words over alphabet Σ^* generated by accepting runs of A.

The language of infinite words accepted by a timed automaton is defined analogously; the relevant acceptance condition is that the underlying infinite run visits locations in F infinitely often. We write $\mathcal{L}_\omega(A)$ to denote the set of infinite timed words accepted by A.

29.5.2 Timed Automata with ε-Transitions

Silent transitions are transitions of the form $(q, g, \varepsilon, r, q')$, where ε is the empty word. In other terms, they are transitions carrying no letter. Silent transitions offer a convenient way of modeling, e.g., internal actions. In the setting of finite-state automata, it is well known that such transitions can be removed, by merging them with the possible subsequent actions.

The question whether the above result extends to the timed setting was settled in [42], with a negative answer: to see this, simply consider the automaton in Fig. 7; its language $\mathcal{L}(A)$ contains precisely those timed words in which all timestamps

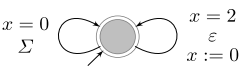


Fig. 7 A timed automaton with ε-transitions

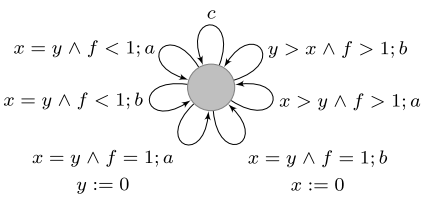


Fig. 8 Automaton accepting words with finitely many a's or finitely many b's

are even integer numbers. Towards a contradiction, assume that there exists a timed automaton B, without ε-transitions, such that $\mathcal{L}(A) = \mathcal{L}(B)$; write m for the maximal integer constant appearing in the timing constraints of B. Then the one-letter word $(\sigma, 2m)$ is accepted by B, since it is accepted by A. Since B has no silent transition, it must have a σ-transition from an initial state to an accepting one. The guard on this transition can only involve constants less than or equal to m, so that B must accept (σ, k) for all $k > m$, which is a contradiction.

Theorem 5 ([42]) *Silent transitions strictly increase the expressive power of timed automata.*

It can be proved that in the case when ε-transitions do not reset any clock, they do not add expressiveness. Finally, let us mention that the question whether a timed automaton with silent transitions is equivalent (i.e., accepts the same language) to some timed automaton without such transitions is undecidable [56].

In the sequel, we consider timed automata without ε-transitions.

29.5.3 Clock Constraints as Acceptance Conditions

Clock constraints can be used to enable or disable certain conditions along the runs of a timed automaton. As such, they can be used to define acceptance conditions, when added on top of a finite-state automaton.

In this setting, we consider the *untimed language* of timed automata: given a timed automaton A, its untimed language (of infinite words) is the set $\mathcal{L}_u$ containing exactly those words $(a_i)_{i\in\mathbb{N}}$ for which there is a *diverging* real-valued sequence $(d_i)_{i\in\mathbb{N}}$ such that the timed word $(a_i, d_i)_{i\in\mathbb{N}} \in \mathcal{L}_\omega(A)$. Notice that thanks to the time-abstracted bisimulation between a timed automaton and its region automaton, the untimed language of a timed automaton is easily seen to be ω-regular.

Conversely, any ω-regular language is the untimed language of a timed automaton: as an example, consider the language of infinite words over $\{a, b, c\}$ that have finitely many a's or finitely many b's. The untimed language of the automaton depicted in Fig. 8 precisely corresponds to that language: indeed, the a- and b-transitions on the left can only be taken finitely many times, since we require time

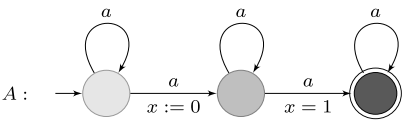


Fig. 9 A non-complementable timed automaton

divergence. Hence one of the a- and the b-transitions at the bottom has to be taken. But after this time, only the corresponding transition on the right is allowed (together with the ε-transition, which is always allowed). This construction can be generalized, so that:

Theorem 6 ([101]) *Given an ω-regular language L, there exists a (one-location) timed automaton A such that $\mathcal{L}_u(A) = L$.*

The number of clocks and locations can be shown to define strict hierarchies of (untimed) ω-regular languages.

More generally, given a finite-state automaton A and an ω-regular language L, one can equip A with clocks and clock constraints in such a way that the untimed language of the resulting timed automaton is the intersection of the language of A with L [101].

29.5.4 Intersection, Union, and Complement

A (finite or infinite) language is said to be *timed regular* if it is accepted by some timed automaton. Timed regular languages (both finite and infinite) are effectively closed under intersection and union. They are however not closed under complement. We reproduce in Fig. 9 an example (taken from [13]) of a timed automaton A, equipped with a single clock, that cannot be complemented: there does not exist a timed automaton A' such that $\mathcal{L}_\omega(A')$ is the set of all timed words not accepted by A. The complement of $\mathcal{L}_\omega(A)$ contains all timed traces in which no pair of a's is separated by exactly one time unit. Intuitively, since there is no bound on the number of a's that can occur in any unit-duration time interval, any timed automaton capturing the complement of $\mathcal{L}_\omega(A)$ would require an unbounded number of clocks to keep track of the times of all the a's within the past one time unit. A formal proof that A cannot be complemented is given in [105].

Under some restrictions, timed automata can be made determinizable (hence also complementable). Most notably, *event-clock automata* [16] enjoy this property. In such timed automata, each letter a of the alphabet is associated with two clocks x_a and y_a (and any clock is associated with some letter that way): clock x_a (called the *event-recording clock* of a) is used to measure the delay elapsed since the last reset of event a (and is initially set to some special value $+\infty$), while y_a (the *event-predicting clock* of a) is used to constrain the delay until the next occurrence of a. One can easily show that event-clock automata can be represented as classical timed automata, though several clocks might be needed to encode each

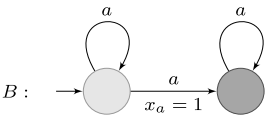


Fig. 10 An event-clock automaton

event-recording clock. Figure 10 displays an example of an event-clock automaton accepting those timed words containing two *consecutive* a's separated by exactly one time unit.

It must be remarked that, at any time during a run of an event-clock automaton on some timed word w, the valuation of the clocks does not depend on the run, but only on w. As a consequence, the classical subset construction for determinising finite-state automata can be adapted to handle event-clock automata, which thus form an (effectively) determinizable and complementable class of timed automata.

29.5.5 Language Emptiness, Inclusion

It follows immediately from Theorem 1 that the language-emptiness problem for timed automata is PSPACE-complete [13]. Unfortunately the language-universality, language-inclusion and language-equivalence problems for timed automata are all undecidable. By contrast, recall from Theorem 4 that the related branching-time counterparts to language inclusion and equivalence, namely similarity and bisimilarity, are both decidable on timed automata.

Theorem 7 ([13]) *The language-inclusion problem for timed automata is undecidable, both over finite and infinite words.*

The proof of Theorem 7 is by reduction from the Halting Problem for Turing machines. This reduction involves encoding the valid halting computations of a given Turing machine M as a timed language whose complement is recognized by a timed automaton A_M which can be effectively computed from M. Intuitively, discrete computation steps of M are simulated over unit-duration time intervals, with timed events used to encode the tape's contents. The integrity of the tape in a valid encoding of a computation is maintained by requiring that any given timed event be preceded and followed at a distance of exactly one time unit by the same timed event, and vice-versa (unless the corresponding character is to be modified by M in the computation step). Figure 11 illustrates this encoding. Note that the density of time enables one to accommodate arbitrarily large tape contents. The key idea is that while no timed automaton can in general accurately capture the encodings of valid computations of a Turing machine, A_M can be engineered to recognise precisely all the *invalid* computations of M; indeed, a computation is invalid if it fails one of finitely many rules, the most interesting of which is to adequately preserve the tape's contents. The latter is easily detected, either by a timed automaton witnessing a timed event with no predecessor one time unit earlier (corresponding to an

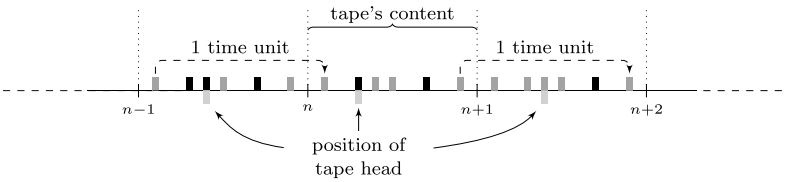


Fig. 11 Encoding computations of Turing machines as timed words

insertion error on the tape), or conversely by a timed automaton witnessing a timed event with no successor one time unit later (corresponding to a *deletion* error on the tape). Other rule failures can likewise be detected by small timed automata. Automaton A_M is obtained as the disjunction of those finitely many timed automata. The upshot is that M fails to have a valid halting computation iff A_M accepts every single timed trace. This shows that universality, and *a fortiori* language inclusion, are indeed undecidable for timed automata.

Theorem 7 places a serious limitation on the algorithmic analysis of timed automata since many verification questions naturally reduce to checking language inclusion. In spite of this hindrance there has been a great deal of research on various aspects of timed language inclusion, including [16, 102, 134] among many others. Here we describe several approaches, involving syntactic and semantic restrictions on timed automata, to obtaining positive decidability results for language inclusion.

Let us first notice that the classical approach to deciding language inclusion is by testing emptiness of the intersection of the first language with the complement of the second one. Thus whether $\mathcal{L}_\omega(A) \subseteq \mathcal{L}_\omega(B)$ is decidable as soon as B can be complemented. For instance:

Theorem 8 ([15]) *Given timed automata A and B, the language-inclusion problem $\mathcal{L}_\omega(A) \subseteq \mathcal{L}_\omega(B)$ is decidable if B is an event-clock automaton.*

Using more elaborate techniques (based on *well-quasi-orderings* and Higman's Lemma), we can prove:

Theorem 9 ([124, 134, 135]) *Given timed automata A and B, the finite-word language-inclusion problem $\mathcal{L}(A) \subseteq \mathcal{L}(B)$ is decidable and non-primitive recursive provided B has at most one clock.*

In the case of infinite words, the language-inclusion problem $\mathcal{L}_\omega(A) \subseteq \mathcal{L}_\omega(B)$ is undecidable even when B has only one clock. The proof of undecidability is by reduction from the boundedness problem for lossy channel machines [4].

A natural semantic restriction on timed automata to recover decidability of language inclusion involves adopting a discrete-time model. Given a timed language $\mathcal{L}$, let $\mathbb{Z}(\mathcal{L})$ denote the set of timed words $(a_1, t_1) \dots (a_p, t_p)$ such that each timestamp t_i lies in $\mathbb{Z}$. Given timed automata A and B, the discrete-time language-inclusion problem is to decide whether $\mathbb{Z}(\mathcal{L}(A)) \subseteq \mathbb{Z}(\mathcal{L}(B))$. This prob-

lem is EXPSPACE-complete: the exponential blow-up over the complexity of the language-inclusion problem for classical non-deterministic finite automata arises from the succinct binary representation of clock values in timed automata. Hardness in EXPSPACE is proven in [28].

Using a technique called *digitization* [102] the discrete behaviors of timed automata can be used to infer conclusions about their dense-time behavior. For example:

Theorem 10 ([102]) *Let A be a closed timed automaton* (*i.e.*, *having only non-strict inequalities as clock constraints*) *and B an open timed automaton* (*i.e.*, *having only strict inequalities*). *Then $\mathcal{L}(A) \subseteq \mathcal{L}(B)$ if and only if $\mathbb{Z}(\mathcal{L}(A)) \subseteq \mathbb{Z}(\mathcal{L}(B))$.*

To apply Theorem 10 one can imagine over-approximating a real-time model by a closed timed automaton and under-approximating a specification by an open timed automaton.

Rather than restricting the precision of the semantics we can instead consider a time-bounded semantics in which we consider only finite timed words of total duration at most N. Note that due to the density of time there is no bound on the number of events that can be performed in a fixed time period. In this case, for the whole class of timed automata, we have:

Theorem 11 ([133]) *Over bounded time* (*i.e.*, *considering only finite timed words of total duration at most N*, *for some fixed time bound N*), *the language-inclusion problem is* 2-EXPSPACE-*complete*.

Theorem 11 was proven as a corollary of the decidability of satisfiability of monadic second-order logic over structures of the form $(I, <, +1)$, with I a bounded interval of reals and $+1$ denoting the plus-one relation: $+1(x, y)$ iff $y = x + 1$.

Notwithstanding the positive decidability results Theorem 9 and Theorem 11, neither one-clock timed automata nor automata over bounded time are closed under complement. (The counterexample in Sect. 29.5.4 can still be used in both cases.) To remedy this deficiency, the strictly more powerful model of *alternating timed automata* has been introduced [124, 135]. Alternating timed automata are a common generalization of timed automata and alternating finite automata; they are closed under all Boolean operations, but language inclusion remains decidable for alternating timed automata with one clock or over bounded time. Unlike in the untimed setting, alternating timed automata are strictly more expressive than purely non-deterministic timed automata. This extra expressiveness is crucially utilised in [135] where it is shown how to translate formulas of Metric Temporal Logic (see Sect. 29.6) into equivalent one-clock alternating timed automata.

29.6 Timed Temporal Logics

The whole theory of temporal-logic model checking has been extended to the setting of timed automata, in order to express and check richer properties beyond emptiness/reachability. We present the most significant results below.

29.6.1 Linear-Time Temporal Logics

The most natural way of extending LTL (see Chap. 2) with quantitative requirements is by decorating modalities with timing constraints. We present the resulting logic, called *Metric Temporal Logic* (MTL), below. Another extension consists in using clocks in formulas, with a way of resetting them when some property is fulfilled and checking their values at a later moment. The resulting logic is called *Timed Propositional Temporal Logic* (TPTL). Due to lack of space, we don't detail the latter logic, and rather refer to [17, 18, 52] for more details about TPTL.

Given a set P of atomic propositions, the formulas of MTL are built from P using Boolean connectives and time-constrained versions of the *until* operator $\mathbf{U}$ as follows:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \mathbf{U}_I \varphi \,,$$

where $I \subseteq (0, \infty)$ is an interval of reals with endpoints in $\mathbb{N} \cup \{\infty\}$. We sometimes abbreviate $\mathbf{U}_{(0,\infty)}$ to $\mathbf{U}$, calling this the *unconstrained* until operator.

Further connectives can be defined following standard conventions. In addition to propositions $\top$ (true) and $\bot$ (false) and disjunction $\vee$, we have the *constrained eventually* operator $\Diamond_I \varphi \equiv \top \mathbf{U}_I \varphi$, and the *constrained always* operator $\Box_I \varphi \equiv \neg\Diamond_I \neg\varphi$.

Sometimes MTL is presented with past connectives (e.g., constrained versions of the "since" connective from LTL) as well as future connectives [17]. However we do not consider past connectives in this chapter.

Next we describe two commonly adopted semantics for MTL.

Continuous Semantics

Given a set of propositions P, a *signal* is a function $f : \mathbb{R}_{\geq 0} \to 2^P$ mapping $t \in \mathbb{R}_{\geq 0}$ to the set $f(t)$ of propositions holding at time t. We say that f has *finite variability* if its set of discontinuities has no accumulation points (in other words, on any finite interval the value of f can only change a finite number of times). In this chapter, we require that all signals be finitely variable. Given an MTL formula φ over the set of propositional variables P, the satisfaction relation $f \models \varphi$ is defined inductively, with the classical rules for atomic propositions and Boolean operators, and with the following rule for the "until" modality, where f^t denotes the signal $f^t(s) = f(t + s)$:

$$f \models \varphi_1 \mathbf{U}_I \varphi_2 \quad \text{iff} \quad \text{for some } t \in I,\ f^t \models \varphi_2 \text{ and } f^u \models \varphi_1 \text{ for all } u \in (0, t).$$

Pointwise Semantics

In the *pointwise semantics* MTL formulas are interpreted over timed words. Given a (finite or infinite) timed word $w = (a_1, t_1), \ldots, (a_n, t_n)$ over alphabet 2^P and an MTL formula φ, the satisfaction relation $w, i \models \varphi$ (read "w satisfies φ at position i") is defined inductively, with the classical rules for Boolean operators, and with the following rule for the "until" modality:

$$w, i \models \varphi_1 \mathbf{U}_I \varphi_2 \quad \text{iff} \quad \text{there exists } j \text{ such that } i < j < |w|, w, j \models \varphi_2, \; t_j - t_i \in I, \text{ and } w, k \models \varphi_1 \text{ for all } k \text{ with } i < k < j.$$

The pointwise semantics is less natural if one thinks of temporal logics as encoding fragments of monadic logic over the reals. On the other hand it seems more suitable when considering MTL formulas as specifications on timed automata. In this vein, when adopting the pointwise semantics it is natural to think of atomic propositions in MTL as referring to events (corresponding to location changes) rather than to locations themselves.

Consider our example of Fig. 2. Using LTL, we can express the property that Process P_i will try to get the resource infinitely many times, by writing $\Box\Diamond\texttt{start!}$. With MTL, we can be more precise and write $\Box\Diamond_{\leq 4}\texttt{start!}$, stating that whatever the current state, within four time units Process P_1 will start trying to acquire the resource. MTL can also be used to express bounded-time response properties, such as $\Box(\texttt{start!} \Rightarrow \Diamond_{\leq 10}\texttt{acquire!})$.

29.6.2 Verification of Linear-Time Temporal Logics

Model checking timed automata can be carried out under either pointwise or continuous semantics. For the latter, it is necessary to alter our definitions to associate a language of *signals* with a timed automaton rather than a language of timed words. In turn, this requires a notion of timed automata in which locations are labelled by atomic propositions. A full development of this semantics can be found, e.g., in [14].

Theorem 12 ([13]) *Model checking and satisfiability for* LTL, *over both the pointwise and continuous semantics, are* PSPACE*-complete.*

The PSPACE upper bound in Theorem 12 can be established in the same manner as in the untimed case, by translating the negated formula to a Büchi automaton, and performing an on-the-fly reachability check on the product of this automaton with the region graph of the model.

Theorem 13 ([135–137]) *Model checking and satisfiability for* MTL *in the pointwise semantics over finite words are decidable but non-primitive recursive. Over infinite words, both problems are undecidable.*

As explained in Sect. 29.5.5, the decidability results are essentially obtained by translating MTL formulas into one-clock alternating timed automata, and rephrasing the model-checking or satisfiability problems as instances of language emptiness in one-clock alternating timed automata.

The undecidability result proceeds by reduction from the recurrent reachability problem for channel machines with insertion errors: the infinite runs of such a machine can be encoded as timed words, which in turn are easily characterized by an MTL formula [136].

Theorem 14 ([14]) *Model checking and satisfiability for* MTL *in the continuous semantics* (*over both finite and infinite signals*) *are undecidable.*

The extra expressiveness of the continuous semantics enables a more direct proof of undecidability than in Theorem 13. In this case, one can directly encode the computations of a Turing machine as timed signals, which again can be captured by an MTL formula, following a scheme similar to that of Theorem 7.

A key ingredient of the undecidability proof of Theorem 14, as well as the non-primitive recursive complexity in Theorem 13, is the ability of MTL to express *punctuality*, i.e., the requirement that two events be separated by an exact duration. A fragment of MTL that syntactically disallows punctuality, known as *Metric Interval Temporal Logic* (MITL), was proposed by Alur *et al.* in [14]. In MITL, one requires that all instances of the interval I appearing in uses of the constrained until operator $\mathbf{U}_I$ be non-singular. The main result of [14] is as follows:

Theorem 15 ([14]) *For both the pointwise and continuous semantics*, *model checking and satisfiability for* MITL *are* EXPSPACE*-complete*, *over both finite and infinite behaviors.*

This theorem was obtained by translating MITLformulas into equivalent timed automata of potentially exponential size. In contrast, it is easy to write an MTL formula that has no equivalent timed automaton: for example, the formula $\neg\Diamond(a \wedge \Diamond_{=1} a)$ captures the complement of the language of automaton A in Fig. 9.

Another decidable fragment of MTL can be obtained by adapting the idea of *event clocks* to temporal logics [140]: here, timing constraints can only refer to the *next* (or *previous*) occurrence of an event. Hence ECTL (standing for *Event-Clock Temporal Logic*) extends LTL with $\triangleright_I \varphi$ and $\triangleleft_I \varphi$. For instance, that there are two *consecutive* a's separated by one time unit is written in ECTL as $\Diamond(a \wedge \triangleright_{=1} a)$.

Theorem 16 ([140]) *Satisfiability and model checking are* PSPACE*-complete for* ECTL, *in either the pointwise or continuous semantics.*

Not surprisingly, deciding ECTLis achieved by a translation to event-clock automata (see Sect. 29.5.4). However, event-clock automata are not powerful enough to precisely capture ECTL, and require the use of *timed Hintikka sequences*. Intuitively, given a formula φ in ECTL, a timed Hintikka sequence is a timed word on

sets of subformulas of φ, required to satisfy local consistency conditions. Compare to a timed word, a timed Hintikka sequence contains more information about the truth value of the subformulas of φ, which will help the event-clock automaton decide whether the underlying timed word is to be accepted. We refer to [140] for more details, and to [104] for an extension of event-clock automata that encompasses ECTL.

Rather than imposing syntactical restrictions, an alternative approach to recovering decidability is to consider a *time-bounded* semantics, i.e., in which either timed words or signals are observed over a fixed, bounded time interval:

Theorem 17 ([133]) *Model checking and satisfiability for* MTL *over bounded time, in either the pointwise or continuous semantics, are* EXPSPACE*-complete.*

The main technique used in the proof of Theorem 17 is an exponential transformation from MTL formulas, given a fixed time bound, into equisatisfiable LTL formulas.

29.6.3 Branching-Time Temporal Logics

Given a set P of atomic propositions, TCTL* formulas are state-formulas obtained as formulas φ_s from the following grammar:

$$\varphi_s ::= p \mid \neg\varphi_s \mid \varphi_s \wedge \varphi_s \mid \mathbf{E}\varphi_p \mid \mathbf{A}\varphi_p$$
$$\varphi_p ::= \varphi_s \mid \neg\varphi_p \mid \varphi_p \wedge \varphi_p \mid \varphi_p \mathbf{U}_I \varphi_p$$

Compared to CTL*, TCTL* has a *time-constrained* until, requiring as for MTL that the right-hand side formula should be fulfilled within that time. The same shorthands as for MTL can be defined, such as $\Diamond_I \varphi$ or $\Box_I \varphi$.

As for the linear-time temporal logics, the semantics of TCTL* comes in (at least) two flavours: continuous and pointwise. The continuous semantics is defined on *dense trees*, which naturally extend classical discrete trees to the continuous setting [7, 89] and represent the set of signals of timed automata; this semantics extends the continuous-time semantics of MTL with path quantifiers.

The pointwise semantics is defined over discrete (but infinite-branching) trees, which can be used to represent the timed words generated by timed automata. This corresponds to evaluating TCTL* formulas over the operational semantics of timed automata, as defined in Sect. 29.2.

Finally, as for the untimed case, the fragment of TCTL* where each temporal modality is under the immediate scope of a path quantifier is of particular interest, and will be called TCTL.

Before turning to the algorithmic part, let us show how TCTLcan be used to express desirable properties of the timed system modeled in Fig. 2. Mutual exclusion (in a setting with two processes P_1 and P_2 and one resource R_1) is expressed as

$\neg \mathbf{E}\Diamond(\text{working}_1 \wedge \text{working}_2)$: two processes will never be working (i.e., using the resource) at the same time. We can also express timing requirements, such as the fact that Process P_i will never be working continuously for more than one time unit: $\mathbf{A}\Box(\mathbf{A}\Diamond_{\leq 1} \neg \text{working}_i)$.

29.6.4 Verification of Branching-Time Temporal Logics

Since TCTL* embeds MTL, there is no hope that its model checking and satisfiability will be decidable. On the lower side, CTL model checking is clearly decidable over timed automata: CTL is invariant under bisimulation, so that any property to be checked on a timed automaton can equivalently be checked on its corresponding finite-state region automaton.

Concerning TCTL, model checking can be shown to be decidable. Actually, this can easily be shown on the explicit-clock version of TCTL (using formula clocks), which strictly subsumes TCTL. For this logic, the important property is that any two region-equivalent states satisfy the same formulas: this can be proven by induction on the structure of the formula. Consider for instance the formula $\zeta = \mathbf{E}\varphi\mathbf{U}_I\psi$, assuming that φ and ψ are compatible with region (i.e., if they hold true in some state, then they also hold true in any region-equivalent state). Given a timed automaton A, consider the automaton A_t obtained by adding an extra clock t to A, which does not modify its behavior. Then if $(\ell, v) \models \zeta$ in A, then $(\ell, v_t) \models \mathbf{E}\varphi\mathbf{U}(\psi \wedge t \in I)$ where v_t extends v by mapping clock t to zero and "$t \in I$" (which is not a TCTL formula but whose meaning is rather clear) is also compatible with regions. In the end, the set of states in A_t where formula $\mathbf{E}\varphi\mathbf{U}(\psi \wedge t \in I)$ is a union of regions, so that it is also the case for the set of states of A where ζ holds.

Using this result, TCTL model checking can be achieved by labeling states of the region automaton with the subformulas they satisfy. This applies for both semantics, with slight differences. It should be noticed that this extends to the explicit-clock version of TCTL, and even to the fragment of explicit-clock TCTL* where formula-clocks are only reset at the level of path quantifiers [66].

While labeling the region automaton requires exponential space, the algorithm can be implemented in a space-efficient manner so as to only use polynomial space. Reachability being already PSPACE-hard, we get the following theorem:

Theorem 18 ([9]) TCTL *model checking is* PSPACE*-complete.*

TCTL (as well as CTL) suffers from not being able to express useful properties, in particular fairness (see Chap. 2 on temporal logics). One way to solve this problem is by decorating path quantifiers with fairness requirements [7, 152]. One can then apply classical algorithms for CTL with fairness [75] or adapt fixpoint characterizations. Another approach is to consider TCTL defined with formula clocks as sketched above, and to have it include CTL*. The resulting logic is very expressive while still enjoying a PSPACE model-checking algorithm [66].

These positive results about model checking do not extend to satisfiability:

Theorem 19 ([9]) TCTL *satisfiability is undecidable.*

The proof follows the same ideas as for undecidability of MTLsatisfiability, by associating universal path-quantifiers with each temporal modality. As for the linear-time case, it suffices to ban equality constraints to recover decidability [117, 118]. This can be proved by lifting tree-automata techniques to the timed setting.

29.7 Symbolic Algorithms, Data Structures, Tools

29.7.1 Zones and Operations

As shown in the previous sections, the regions introduced in Sect. 29.3 provide a finite and elegant abstraction of the infinite state space of timed automata, enabling us to prove decidability of a wide range of problems, including (timed and untimed) bisimilarity, untimed language equivalence and language emptiness, as well as TCTL model checking.

Unfortunately, the number of states obtained from the region partitioning is extremely large. Indeed, it is exponential in the number of clocks as well as in (the binary representation of) the maximal constants of the timed automaton [13]. Efforts have been made to develop more efficient representations of the state space [34, 39, 103, 121], using the notion of *zones* introduced below as a coarser and more compact representation of the state space.

For a finite set C of clocks, a subset $Z \subseteq \mathbb{R}^C_{\geq 0}$ is called a *zone* if there exists $\varphi \in \Phi_d(C)$ for which $Z = [\![\varphi]\!]_C$. For reachability analysis, we need the following operations on zones: for a zone $Z \subseteq \mathbb{R}^C_{\geq 0}$ and $r \subseteq C$, let us denote

- the *delay* of Z by $Z^{\uparrow} = \{v + d \mid v \in Z, d \in \mathbb{R}_{\geq 0}\}$ and
- the *reset* of Z under r by $Z[r] = \{v[r] \mid v \in Z\}$.

Lemma 1 ([103, 157]) *Let Z, Z' be zones over C and $r \subseteq C$. Then $Z^{\uparrow}$, $Z[r]$, and $Z \cap Z'$ are also zones over C.*

Definition 5 The *zone automaton* associated with a timed automaton $A = (L, \ell_0, C, \Sigma, I, E)$ is the transition system $[\![A]\!]_Z = (S, s_0, \Sigma \cup \{\delta\}, T)$ given as follows:

$$S = \big\{(\ell, Z) \mid \ell \in L, Z \subseteq \mathbb{R}^C_{\geq 0} \text{ is a zone}\big\} \qquad s_0 = \big(\ell_0, [\![v_0]\!]\big)$$

$$T = \big\{(\ell, Z) \overset{\delta}{\rightsquigarrow} \big(\ell, Z^{\uparrow} \cap [\![I(\ell)]\!]_C\big)\big\}$$
$$\cup \big\{(\ell, Z) \overset{a}{\rightsquigarrow} \big(\ell', (Z \cap [\![\varphi]\!]_C)[r] \cap [\![I(\ell')]\!]_C\big) \mid \ell \xrightarrow{\varphi, a, r} \ell' \in E\big\}$$

Analogously to Proposition 1, we have:

Proposition 2 ([157]) *A location ℓ in a timed automaton $A = (L, \ell_0, F, C, \Sigma, I, E)$ is reachable if and only if there is a zone $Z \subseteq \mathbb{R}^C_{\geq 0}$ for which (ℓ, Z) is reachable in $[\![A]\!]_Z$.*

A priori, however, the zone automaton defined above is infinite, hence another, finite, abstraction is needed. This is provided by *normalization* using region equivalence $\cong_M$: for a maximal constant M, the *normalization* of a zone $Z \subseteq \mathbb{R}^C_{\geq 0}$ is the set $\{v : C \to \mathbb{R}_{\geq 0} \mid \exists v' \in Z : v \cong_M v'\}$. The normalization of a zone is not in general a zone, hence in practice other normalization operators are used (see Sect. 29.7.3).

The normalized zone automaton is defined analogously to the zone automaton defined above, and in case the timed automaton to be verified does not contain diagonal clock constraints of the form $x - y \bowtie k$, Proposition 2 also holds for the normalized zone automaton. Hence we can obtain a reachability algorithm by applying any search strategy (depth-first, breadth-first, or another) on the normalized zone automaton.

For timed automata that contain diagonal clock constraints $x - y \bowtie k$, however, it can be shown [38, 48] that normalization as defined above does *not* give rise to a sound and complete characterization of forward reachability. Instead, one can apply a refined normalization which depends on the difference constraints used in the timed automaton, see [38].

29.7.2 Symbolic Datastructures

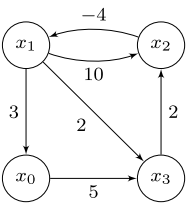
A zone, given by a conjunction of elementary clock constraints, may be represented using a directed weighted graph, where the nodes correspond to the clocks in C together with an extra "zero" clock x_0, and an edge $x_i \xrightarrow{k} x_j$ corresponds to a constraint $x_i - x_j \leq k$ (if there is more than one upper bound on $x_i - x_j$, k is the minimum of all these constraints' right-hand sides). The extra clock x_0 is fixed at value 0, so that a constraint $x_i \leq k$ can be represented as $x_i - x_0 \leq k$. Lower bounds on $x_i - x_j$ are represented as (possibly negative) upper bounds on $x_j - x_i$, and strict bounds $x_i - x_j < k$ are represented by adding a flag to the corresponding edge.

The weighted graph in turn may be represented by its adjacency matrix, which in this context is known as a *difference-bound matrix* or DBM. The above technique was introduced in [87] (the main ideas were already present in [43]). Figure 12 gives an illustration of an extended clock constraint together with its representation as a difference-bound matrix. Note that the clock constraint contains superfluous information.

On zones represented using DBMs, efficient (in time cubic in the number of clocks in C) algorithms are available for computing delays, resets and intersections. For reachability checking, other necessary operations inclusion checking whether $Z \subseteq Z'$, and emptiness checking, $Z = \emptyset$; these can also be computed efficiently using DBMs.

Fig. 12 Graph representation of extended clock constraint

$$Z = \begin{cases} x_1 \leq 3 \\ x_1 - x_2 \leq 10 \\ x_1 - x_2 \geq 4 \\ x_1 - x_3 \leq 2 \\ x_3 - x_2 \leq 2 \\ x_3 \geq -5 \end{cases}$$

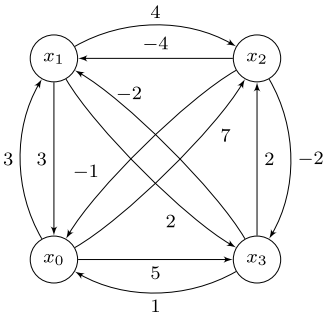


For these computations, a canonical representation obtained as the *shortest-path closure* of the DBM graph is used. Another useful canonical form is the *shortest-path reduction* described in [120]. Whereas the shortest-path closure form gives all derived constraints, the shortest-path reduced form provides a memory-efficient representation, containing only a minimal set of constraints. Figure 13 shows the two canonical forms of the DBM of Fig. 12. Given the shortest-path closure of the DBM graph, the shortest-path reduced form is obtained by partitioning the set of clocks according to zero cycles in the DBM graph; in the DBM of Fig. 13, $\{x_1, x_2, x_3\}$ constitutes one such class, and $\{x_0\}$ is another one. The reduced form is now obtained by maintaining a minimal set of constraints for each class, essentially a simple cycle of the clocks of the class, and only keeping constraints between representatives of different classes, here x_1 and x_0.

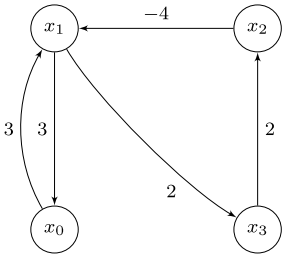
To combat state-space explosion in zone graphs, several optimizations are used. One such approach is to detect whether a state (ℓ, Z) reached through the algorithm is contained in another state (ℓ, Z') which has already been explored. In this case, exploration of (ℓ, Z) will be unnecessary.

Another such optimization is to work with *unions of zones*. If states (ℓ, Z) and (ℓ, Z') are found to reachable during the analysis, then we know that altogether the state $(\ell, Z \cup Z')$ is reachable. Unfortunately, unions of zones are not generally themselves zones, so cannot be efficiently represented using DBMs.

For representing unions of zones, or *federations* as they are called in this context, a data structure inspired by decision diagrams called *clock difference diagrams* or CDDs is used [121]. However, no efficient algorithms are known to compute delays



Shortest-path closure



Shortest-path reduction

Fig. 13 Canonical representations

or resets of federations using CDDs, so in practice reachability analysis using CDDs is done by extracting the zones from the CDD and performing the operations on them one by one.

Other promising data structures in this context are *numerical decision diagrams* [24], *difference decision diagrams* [132], *clock-restriction diagrams* [154], *max-plus polyhedra* [5], *constraint matrix diagrams* [88], and *time-darts* [111]; generally, the design of efficient data structures for symbolic exploration of timed automata is a field of active research.

29.7.3 Practical Efficiency

Symbolic, zone-based exploration of the (reachable) state-space of timed automata using the DBM data structure is key to their analysis. However, a number of additional algorithmic techniques have been developed for gaining efficiency in practice. In the following we indicate a number of these.

As described in Sect. 29.7.1, normalization of zones with respect to the maximum constant M appearing in the given timed automata ensures finiteness of the zone graph and hence termination of algorithms searching the zone graph. Here, a practical problem is that the normalization of a zone is in general not a zone itself, but rather a finite union of such. However, given a representation of the zone as a *shortest-path-closed* DBM, a syntactic *extrapolation* operation that removes bounds that are larger than the maximum constant may be easily performed: any upper bound constraint of the form $x_i - x_j < k$ where $k > M$ is removed, and any lower bound of the form $x_i - x_j > k$ where $k < -M$ is replaced by $x_i - x_j > -M$. Clearly, under extrapolation, only a finite number of DBMs will be encountered, ensuring termination. Furthermore, the correctness is based on the fact that extrapolation of a zone (based on its DBM representation) is included in its normalization, as shown in [29, 48].

Coarser, yet complete, notions of extrapolation have been obtained by performing the operation with respect to *clock- and location-dependent* maximum constants [29] and further differentiating the maximum constant used in upper or lower bound comparisons [30, 31]. In fact, this last extrapolation yields performance comparable to that of the overapproximate *convex hull* abstraction [81].

Several of the algorithmic problems presented in this chapter—e.g., reachability, model checking, and equivalence checking, as well as notions of optimal reachability and controllability, which will be described in later sections—have a fixed-point characterization. Though easy to implement, this leads to backwards iterative algorithms, requiring one to consider and classify states which are possibly not even reachable. Thus, for most problems, so-called *on-the-fly* algorithms have been devised, where the satisfaction of the given property by the initial state is attempted to be concluded in as *local* a fashion as possible, only exploring the state space when needed. For the analysis of timed-automata-based models, such on-the-fly algorithms have been proposed for instance for reachability [120, 157], live-

ness checking [147, 151], model checking with respect to TCTL [47], time-abstract bisimulation checking [149], and controller synthesis for timed game automata [69].

Despite the above efforts in applying aggressive (yet complete) abstractions, the analysis of timed-automata-based models suffers from the state-space explosion problem. Thus, complementary techniques have been proposed and implemented for reducing space consumption at the expense of time performance [35, 120]. Also, various AI-inspired techniques have been developed for *efficient guidance* of the symbolic exploration of timed automata towards specified error states [114–116]. Similarly, the technique of *symmetry reduction* has been developed and implemented for networks of timed automata with several symmetric components, potentially yielding an exponential gain in performance [96]. Moreover, so-called *time-convexity* analysis provides significant performance improvement [153].

Finally, several attempts have been made to extend the technique of *partial-order reduction* to networks of timed automata. In contrast to the proved effect in the finite-state setting, early attempts [37, 131] did not show any improvement in performance compared with regular symbolic exploration. In fact, being a strong synchronizer in a timed-automata network, time reduces the number of independent transitions, which are key for partial-order reduction to have effect. In later work [36, 125, 126], it was found that the union of all zones reached by different interleavings of the same set of transitions is convex, providing the basis of substantial improvement.

Also, bounded model-checking techniques have been developed for timed automata using *difference logics*, though with a limited performance improvement [76, 77].

29.7.4 Tools and Applications

Timed automata and their extensions have been applied to the modeling, analysis and optimization of numerous real-time applications. In this section we give a few examples, not aiming at being exhaustive but rather to illustrate the wide range of application domains.

A variety of mature tools are available which provide important computer-aided support for applications. Well-known tools include UPPAAL [122], KRONOS [158], RED [155] and HyTech [99]. A larger number of other tools related to the analysis of timed automata have emerged over the years including Else [159], Rabbit [44], Verics [84], and TAME [22] as well as tools for analyzing other timed formalisms based on translation to timed automata including Times [21] (task automata), Moby [145] (PLC programs), SART [45] (Safety Critical Java), ART [93] (task graphs), Romeo [91] and TAPALL [68] (Time and Timed-arc Petri Nets), and VeSTA [112] (component integration checking).

The timed-automata formalism is now routinely applied to the modeling and analysis of real-time control programs, including a wide class of Programmable Logic Controller (PLC) control programs [85, 127] and timing analysis and code

generation of vehicle control software [150]. The timed-automata approach has also demonstrated its viability in the timing analysis of certain classes of asynchronous circuits [60].

Similarly, numerous real-time communication protocols have been analyzed using timed automata technology, often with inconsistencies being revealed: e.g., using real-time model checking, the cause of a ten-year-old bug in the IR-link protocol used by Bang & Olufsen was identified and corrected [95]. Most recently, real-time model checking has been applied to the clock synchronization algorithm currently used in a wireless sensor network that has been developed by the Dutch company CHESS [144]. Here it is shown that in certain cases a static, fully synchronized network may eventually become unsynchronized if the current algorithm is used, even in a setting with infinitesimal clock drifts.

During the last few years, timed automata modeling of multitasking applications running under real-time operating systems has received substantial research effort. Here the goals are multiple: to obtain less-pessimistic worst-case response time analysis compared with classical methods for single-processor systems [156]; to relax the constraints of period task arrival times of classical scheduling theory to task arrival patterns that can be described using timed automata [90]; to allow for schedulability analysis of tasks in terms of concurrent objects executing on multiprocessor or distributed platforms (e.g., MPSoC) [61, 80, 109].

Just as symbolic reachability checking of finite-state models has led to very efficient planning and scheduling algorithms, reachability checking for (priced) timed automata has demonstrated competitive and complementary performance with respect to classical approaches such as MIPL on optimal scheduling problems involving real-time constraints, e.g., job-shop and task-graph scheduling [1, 32] and aircraft landing problems [119]. In fact a translation of the variant PDDL3 of PDDL (Planning Domain Definition Language) into priced timed automata has been made [86] allowing optimal planning questions to be answered by cost-optimal reachability checking. Industrial applications include planning a wafer scanner from the semiconductor industry [97] and computation of optimal paper paths for printers [108].

Most recently, computation of winning strategies for timed games has been applied to controller synthesis for embedded systems, including synthesis of most general non-preemptive online schedulers for real-time systems with sporadic tasks [6], synthesis of climate control for pig shed provided by the company Skov [110], and automatic synthesis of robust and near-optimal controllers for industrial hydraulic pumps [71].

29.8 Weighted Timed Automata

Time is not the only quantity one may want to measure when checking an embedded system: one may need to keep track of the battery charge or of the level of oil in a tank. Hybrid automata [98, 100] extend timed automata with extra variables that can

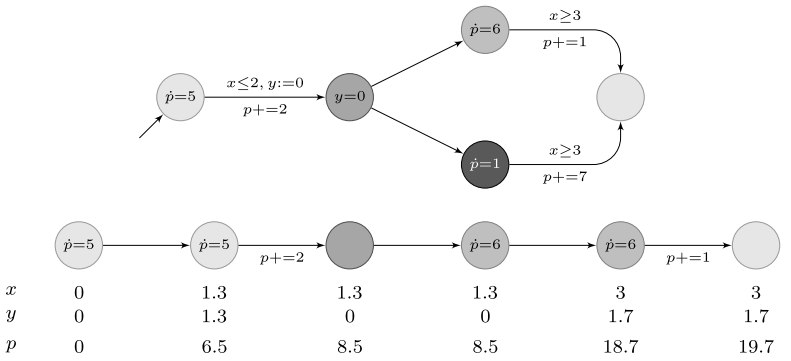


Fig. 14 Example of a weighted timed automaton

help measure such quantities. Unfortunately, reachability is undecidable for these models, even with two-slope hybrid variables. Weighted timed automata [20, 33] is an intermediary model, extending timed automata with hybrid *observer* variables: these variables cannot appear in the guards of the automaton, but they can be used, for example, for optimization purposes. The special case where the observer is a stopwatch (computing the accumulated delay in some locations) was already introduced and solved in [11].

Formally, a *weighted timed automaton* is a pair (A, w) where A is a timed automaton and w labels the locations and edges of A with an integer (or a vector of integers for automata with multiple observer variables). For a transition t, $w(t)$ is the value by which the value of the observer variable is increased, while for a location ℓ, $w(\ell)$ is the *rate* by which the variable increases w.r.t. time (in other words, the observer variable p follows the differential equation $dp/dt = w(\ell)$).

The semantics of a weighted timed automaton (A, w) is that of the underlying timed automaton A. Each run of A is decorated with the value of the observer variable. Figure 14 shows an example of a weighted timed automaton,[3] and a run of this automaton. This run reaches the rightmost location within 3 time units, and with a total weight of 19.7.

29.8.1 Cost-Optimal Schedules

Natural questions on this family of models include optimal reachability of a given location, or optimal mean-cost of infinite runs. Formally, the associated decision problems respectively ask whether the target location can be reached with total weight less than a given threshold, and whether there is an infinite run along which the average weight is less than the given bound.

[3] Notice that the labeling in the second location is a clock invariant (enforcing that no time will elapse in that location). The rate of p is not given in that location as no time will elapse anyway.

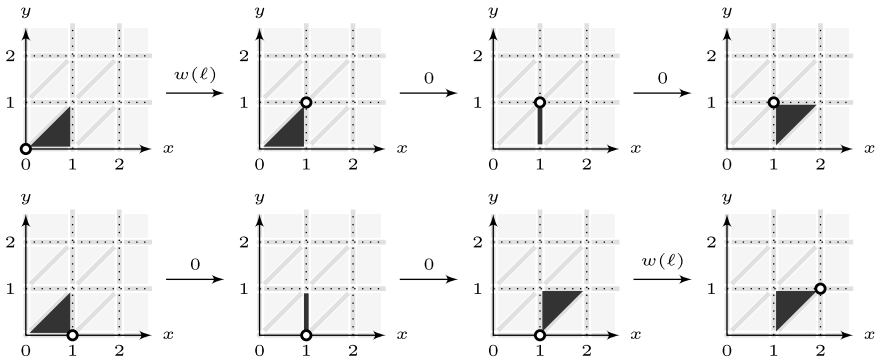


Fig. 15 Two different runs in the corner-point abstraction

These problems turn out to be decidable. The main technique used in the algorithms is a refinement of the region equivalence of Sect. 29.3 called *corner-point equivalence* [50]. Intuitively, for these two optimization problems, optimal schedules will amount to staying *as long as possible* in interesting locations. Corner-point regions extend classical regions with an extra *corner* of this region, i.e., an integer valuation that belongs to the closure of the region. A corner-point (r, c) represents a clock valuation v that is *close* to the corner c but lies within the region r. The *corner-point automaton* is the weighted automaton $\mathcal{C}_{\simeq M} = (S, s_0, \Sigma, T)$ where $S \subseteq L \times \mathcal{R} \times \{0, \ldots, M\}^C$ is the set of states (writing $\mathcal{R}$ for the set of regions), with three kinds of transitions:

- action transitions: there is a transition from (ℓ, R, c) to (ℓ', R', c') if there is a transition $t = (\ell, \varphi, a, r, \ell')$ in A such that $R \subseteq [\![\varphi]\!]_C$, with $R' = R[r]$ and $c' = c[r]$. The weight of this transition is $w(t)$.
- ε-delay transitions: these are transitions from (ℓ, R, c) to (ℓ, R', c), where R' is the immediate time-successor of R sharing corner c. Such a transition corresponds to a very small delay, and its corresponding weight is set to zero.
- 1-delay transitions: these are transitions from (ℓ, R, c) to (ℓ, R, c'), where $c' = c+1$. This corresponds to spending (almost) one time unit in region (ℓ, R). Notice that such a transition is only available if c and $c+1$ are corners of R. The weight of this transition is $w(\ell)$.

Figure 15 displays two sequences of delay transitions in the corner-point automaton; while both sequences depart from the same region and visit the same sequence of locations, the accumulated weight evolves very differently along the two sequences—which explains why we have to refine regions.

The corner-point automaton enjoys the following properties: if there is a run from some location ℓ to some location ℓ' with total weight m in a given weighted timed automaton, then there is a run from $(\ell, \mathbf{0}, \mathbf{0})$ to some (ℓ', R, c) with total weight at most m in the corresponding corner-point automaton; in other words, running through corner-points is always better when trying to optimize the value of the total weight. Conversely, for any positive ε, if there is a run from $(\ell, \mathbf{0}, \mathbf{0})$ to

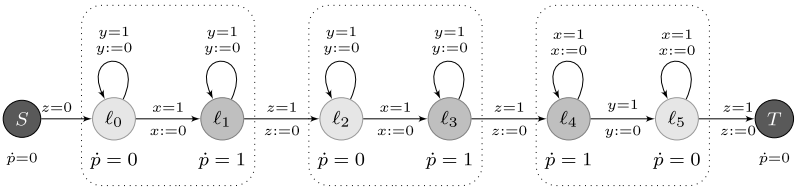


Fig. 16 Module for testing whether $y = 2x$

some (ℓ', R, c) with weight m in the corner-point automaton, then there is a run in the original weighted timed automaton from ℓ (with initial valuation $\mathbf{0}$) to ℓ', with total weight at most $m + \varepsilon$.

This statement can be extended in various ways, so as to handle optimization of the ratio between two variables along finite or infinite runs. In the end:

Theorem 20 ([50]) *The optimal-reachability and the optimal-ratio problems are* PSPACE*-complete.*

Other related problems have been considered in the literature, such as conditional optimal reachability on multi-weighted timed automata [123]. The aim in this setting is to minimize the value of one variable under some conditions on the other variables. We refer to [123], where the problem is shown to be decidable.

29.8.2 Weighted Temporal Logics

Unfortunately, the encouraging results above do not extend to richer properties that could be expressed in weighted extensions of classical temporal logics.[4] While this is not surprising for linear-time temporal logics (as these logics are already mostly undecidable in the timed setting), this also holds for weighted extensions of CTL, be it with modalities decorated with weight constraints (writing $\mathbf{E}\Diamond_{\leq 10} T$ to express that T can be reached within total cost less than 10), or with explicit constraints as atomic formulas (writing $\mathbf{E}\Diamond(T \wedge c \leq 10)$ to express the same property) [49, 63]. Undecidability can be proved by encoding the halting problem for a two-counter machine, where each counter is encoded by a clock of the timed automaton. The central trick in the reduction is the ability to multiply the value of a clock by some constant (while preserving the value of the other clocks). This is achieved by the automaton depicted in Fig. 16, in which we enforce the condition that location T must be reachable from S with total cost exactly 1: indeed, the total cost accumulated from S to T is precisely $1 + 2x_0 - y_0$, where x_0 and y_0 are the values of clocks x

[4] Even if we restrict to nonnegative weights, which is what we assume in this subsection on temporal logics.

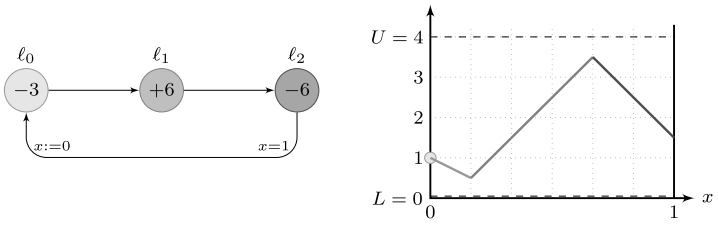


Fig. 17 A weighted timed automaton under energy constraints

and y in S. This provides us with a way of doubling the value of clock x, by letting clock y play the role of x afterwards. Using this module, it is easy to build a complete reduction involving four clocks, which can be further improved to only three clocks.

Theorem 21 ([49, 63]) WCTL *model checking is undecidable* (*on weighted timed automata with at least three clocks*).

Notice that the standard restriction to non-punctual constraints in the logic does not help, as the above reduction can be carried out using only inequality constraints. One way of recovering decidability is to restrict to *one-clock* weighted timed automata. These automata enjoy several special properties which allow us to prove that refining regions to a small granularity (in $O(C^{-h(\varphi)})$ where C is the maximal rate in the automaton and $h(\varphi)$ is the temporal height of the formula being checked) provides a correct finite-state abstraction on which φ can be checked. It follows:

Theorem 22 ([57]) WCTL *model checking is* PSPACE*-complete on one-clock weighted timed automata*.

29.8.3 Energy Constraints

Recently, weighted timed automata have been pushed one step closer to hybrid automata, with the introduction of *energy constraints* [55, 74]. These constraints aim at modeling, for example, autonomous robots, which often must take care of their battery charge level, and ensure that they never run out of energy. This is modeled with weighted timed automata, with the constraint that the accumulated value of the variable must never drop below 0 (or any lower bound). The same problem can of course be considered with an additional upper-bound constraint. Notice that this is a departure from the motto that the cost variable is an observer. Figure 17 displays an example of a weighted timed automaton, together with the evolution of the variable along one particular run. This corresponds to a *feasible* (prefix of a) run, as the variable remains between the lower bound L and upper bound U.

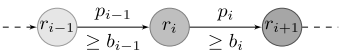


Fig. 18 A *linear* weighted automaton

Only a few results have been obtained so far. Let us begin with the untimed case [55]: for lower-bound constraints, the problem still amounts to optimizing the accumulated cost, with the extra *energy constraint*. In that setting, the Bellman–Ford algorithm can be used to compute the maximal accumulated cost that can be achieved from the initial state, with the extra energy constraint. This provides a polynomial-time algorithm for solving reachability under lower-bound constraints. In case we also have an upper-bound constraint, the problem can be solved in exponential time by augmenting the state space with the explicit value of the variable.

In the timed setting, the only known positive results concern one-clock weighted timed automata under lower-bound constraints [54]. The central technique is the computation of an optimal schedule through a finite *linear* automaton (i.e., visiting all its locations with a fixed, linear order), such as the one depicted in Fig. 18. Notice that along such runs, we allow *lower-bound constraints* (written $\geq b_i$) on each transition. Along such a path, one can prove that the optimal policy is to spend no time in a location r_i if

- either $r_{i-1} > r_i$ (in which case it is more profitable to spend time in location r_{i-1});
- or $r_{i+1} \geq r_i$ and $b_{i-1} + p_{i-1} \geq b_i$ (in which case it is possible to directly jump to the more profitable location r_{i+1}).

If any of these conditions is fulfilled, location r_i can be dropped, and replaced by a transition from r_{i-1} to r_{i+1} with weight $p_{i-1} + p_{i+1}$ and cost-constraint $\geq \max(b_{i-1}, b_i - p_{i-1})$. This provides us with a linear automaton along which the rates are increasing. The optimal policy along such a path can be proved to be to exit a location as soon as the cost constraint $\geq b_i$ is fulfilled. This gives a way of computing the optimal achievable energy level at the end of the path as a function of the initial credit. This extends to one-clock automata by composing such functions. In the end:

Theorem 23 ([54]) *Optimal reachability is decidable in one-clock weighted timed automata under lower-bound constraints.*

Unfortunately, this algorithm does not extend to n-clock automata: indeed, one can easily come up with a small module to increase or decrease the value of the cost variable by the value of a clock (see Fig. 16), thus providing a way of checking linear constraints between several clocks. As a consequence:

Theorem 24 ([58]) *Optimal reachability is undecidable in four-clock weighted timed automata under lower-bound constraints.*

Weighted timed automata under energy constraints are a very recent and active topic with many open problems. Several directions are currently being explored,

such as the extension to *exponential variables*, where the variable follows the differential equation $dp/dt = w(\ell) \cdot p$, or the inclusion of imprecisions in clock values or variable growth.

29.9 Timed Games

Games provide a nice framework for modeling and reasoning about the interactions between various agents (a reactive system and its environment, several components, etc.). We refer to Chap. 27 for details about games and their use for synthesizing correct models.

We consider two-player timed games, in which transitions are partitioned into controllable and uncontrollable (i.e., under the control of an environment). The problem is then to synthesize a strategy telling *when* to take *which* (enabled) controllable transitions in order that a given objective is guaranteed regardless of the behavior of the environment.

Definition 6 A *timed game* is a tuple $G = (L, \ell_0, C, \Sigma_c, \Sigma_u, I, E)$ with $\Sigma_c \cap \Sigma_u = \emptyset$ for which the tuple $A_G = (L, \ell_0, C, \Sigma = \Sigma_c \cup \Sigma_u, I, E)$ is a timed automaton. We require this automaton to be deterministic (so that from any state, an action in Σ corresponds to a unique transition). Edges with actions in Σ_c are said to be *controllable*, those with actions in Σ_u are *uncontrollable*.

We shall again assume a set F of accepting locations to be given for the rest of this section. A *strategy* in a timed game G provides instructions as to which controllable edge to take, or whether to wait, in a given state. Hence it is a mapping σ from finite runs of the underlying timed automaton A_G to $\Sigma_c \cup \{\delta\}$, where $\delta \notin \Sigma$, such that for any run $\rho = (\ell_0, v_0) \to \cdots \to (\ell_k, v_k)$,

- if $\sigma(\rho) = \delta$, then $(\ell, v) \xrightarrow{d} (\ell, v + d)$ is a transition in $[\![A_G]\!]$ for some $d > 0$, and
- if $\sigma(\rho) = a$, then $(\ell, v) \xrightarrow{a} (\ell', v')$ is a transition in $[\![A_G]\!]$.

An *outcome* of a strategy is any run which adheres to its instructions. Such an outcome is said to be *maximal* if it stops in an accepting location, or if no controllable actions are available at its end. An underlying assumption is that uncontrollable actions cannot be forced, hence a maximal outcome which does not end in a final location may "get stuck" in a non-final location. The aim of reachability games is to find strategies all of whose maximal outcomes end in an accepting location; the aim of safety games is to find strategies all of whose (not necessarily maximal) outcomes avoid accepting locations:

Definition 7 A strategy is said to be *winning for the reachability game* if any of its maximal outcomes is an accepting run. It is said to be *winning for the safety game* if none of its outcomes are accepting.

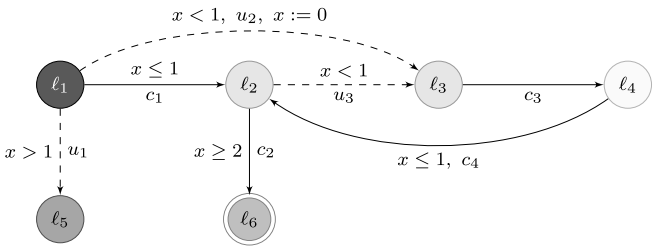


Fig. 19 A timed game with one clock. Controllable edges (with actions from Σ_c) are *solid*, uncontrollable edges (with actions from Σ_u) are *dashed*

Example 5 Figure 19 provides a simple example of a timed game. Here, $\Sigma_c = \{c_1, c_2, c_3, c_4\}$ and $\Sigma_2 = \{u_1, u_2, u_3\}$, and the controllable edges are drawn with solid lines, the uncontrollable ones with dashed lines. The following memoryless strategy is winning for the reachability game:

$$\sigma(\ell_1, v) = \begin{cases} \delta & \text{if } v(x) \neq 1 \\ c_1 & \text{if } v(x) = 1 \end{cases} \qquad \sigma(\ell_2, v) = \begin{cases} \delta & \text{if } v(x) < 2 \\ c_2 & \text{if } v(x) \geq 2 \end{cases}$$

$$\sigma(\ell_3, v) = \begin{cases} \delta & \text{if } v(x) < 1 \\ c_3 & \text{if } v(x) \geq 1 \end{cases} \qquad \sigma(\ell_4, v) = \begin{cases} \delta & \text{if } v(x) \neq 1 \\ c_4 & \text{if } v(x) = 1 \end{cases}$$

Theorem 25 ([25, 26, 65, 128]) *The (time-optimal) reachability and safety games are decidable for timed games. They are* EXPTIME*-complete.*

A key ingredient in the proof of the above theorem is the fact that for reachability as well as safety games, it is sufficient to consider *memoryless* strategies, which only observe the last configuration of a run. This is not the case for other, more subtle, control objectives (e.g., counting properties modulo some N) as well as for the synthesis of winning strategies under *partial observability*. The other ingredient is the region abstraction: if there is a winning strategy, then there is one that only depends on the current region. This provides an exponential-time algorithm, which can be proved to be optimal.

A problem with the above approach is that the safety game can be won by preventing time from diverging. In order to rule out such behaviors, a solution was proposed in [82]; it uses a more symmetric presentation of games, in which both players have a strategy which proposes at the same time the amount of time this player wants to delay, and the transition she wants to take afterwards. At each step, the player with the shortest delay is chosen and her choice is performed. With this definition, if time converges along an outcome, then the player(s) who have applied their choices infinitely many times must have proposed converging sequences of delays. By adding a fairness requirement to the winning condition, we can declare this kind of behavior losing. Deciding the existence of winning strategies for reachability and safety objectives remains EXPTIME-complete in this context.

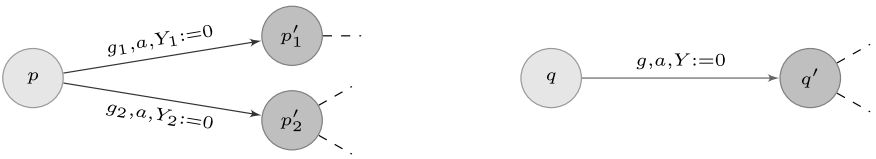


Fig. 20 Two states of a timed automaton

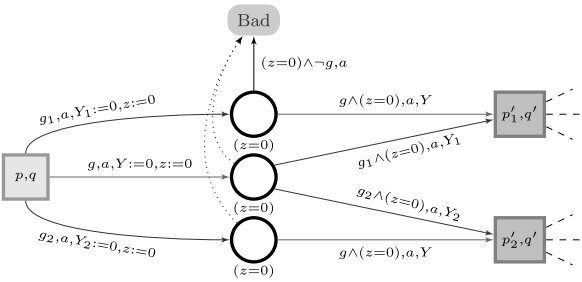


Fig. 21 Timed bisimilarity as a timed game

The field of timed games is a very active research area. From the region-based decidability results, efficient on-the-fly algorithms have been developed [69, 148] and implemented in UPPAAL. In [70] these algorithms have been extended to timed games under partial observability. Research has also been conducted towards the synthesis of *optimal* winning strategies for reachability games on *weighted timed games*. In [8, 51] computability of optimal strategies is shown under a certain condition of *strong cost non-Zenoness*, requiring that the total weight diverges at a given minimum rate per time. Later undecidability results [49, 64] show that for weighted timed games with three or more clocks, this condition (or a similar one) is necessary. It is proved in [59] that optimal reachability strategies are computable for one-clock weighted timed games, though there is an unsettled (large) gap between the known lower bound complexity and an upper bound of 3-EXPTIME, which was recently lowered to EXPTIME [94, 142].

We conclude this section by illustrating how timed games can be used to decide timed bisimilarity of two states of a timed automaton. This provides a simple proof of Theorem 4, which we explain on a small example: consider the states of Fig. 20. That two states (p, v) and (q, w) (where v and w are two valuations of the same set of clocks $\mathcal{C}$) are timed bisimilar means that any transition from either state can be mimicked from the other one, ending up in states that are again bisimilar. We can see this as a game on the product of two copies of the automaton (see Fig. 21): from the joint state $((p, q), (v, w))$, the first player chooses to apply one transition from one of the states (p, v) and (q, w), and the second player has to respond (immediately) with an appropriate move from the other state. The second player has a strategy to always avoid the Bad state if, and only if, the starting states are timed bisimilar. This provides an exponential-time algorithm for checking timed bisimilarity [10].

29.10 Ongoing and Future Challenges

In this chapter, we have surveyed timed automata and the theoretical developments that have led to their being widely accepted as a model for modeling and reasoning about real-time systems. Many developments are still ongoing: we briefly list here some important topics which we think are among the important avenues to be explored during the coming years:

- Robustness (in the timed setting) and implementability [83, 92, 139, 143] address the problem of reconciling the semantics of timed automata (with real-valued clocks) with the models they represent (which usually run at a finite frequency).
- Statistical model checking consists in checking several runs of the model against a given property, and compute statistics to get an estimate of the correctness of the model. The approach has recently been studied and implemented in the setting of stochastic timed automata, where it provides interesting results, even for problems that are otherwise undecidable [79].
- Games on timed automata have received a lot of attention over the last ten years, as they are a convenient formalism for the automated synthesis of real-time systems. Recent extensions to non-zero-sum games [62, 113], where the players have their own objectives, open a rich and promising area of research for synthesizing complex systems.

Acknowledgements We thank the reviewers for their numerous comments, remarks and additional references, which greatly helped us improve this chapter.

References

1. Abdeddaïm, Y., Asarin, E., Maler, O.: Scheduling with timed automata. Theor. Comput. Sci. **354**(2), 272–300 (2006)
2. Abdeddaïm, Y., Maler, O.: Job-shop scheduling using timed automata. In: Berry, G., Comon, H., Finkel, A. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 2102, pp. 478–492. Springer, Heidelberg (2001)
3. Abdeddaïm, Y., Maler, O.: Preemptive job-shop scheduling using stopwatch automata. In: Katoen, J.-P., Stevens, P. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 2280, pp. 113–126. Springer, Heidelberg (2002)
4. Abdulla, P.A., Deneux, J., Ouaknine, J., Worrell, J.: Decidability and complexity results for timed automata via channel machines. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) International Colloquium on Automata, Languages and Programming (ICALP). LNCS, vol. 3580, pp. 1089–1101. Springer, Heidelberg (2005)
5. Allamigeon, X., Gaubert, S., Goubault, É.: Inferring min and max invariants using max-plus polyhedra. In: Alpuente, M., Vidal, G. (eds.) Intl. Symp. on Static Analysis (SAS). LNCS, vol. 5079, pp. 189–204. Springer, Heidelberg (2008)
6. Altisen, K., Gößler, G., Pnueli, A., Sifakis, J., Tripakis, S., Yovine, S.: A framework for scheduler synthesis. In: Symposium on Real-Time Systems (RTSS), pp. 154–163. IEEE, Piscataway (1999)
7. Alur, R.: Techniques for automatic verification of real-time systems. PhD thesis, Stanford University (1991)

8. Alur, R., Bernadsky, M., Madhusudan, P.: Optimal reachability for weighted timed games. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) International Colloquium on Automata, Languages and Programming (ICALP). LNCS, vol. 3142, pp. 122–133. Springer, Heidelberg (2004)
9. Alur, R., Courcoubetis, C., Dill, D.L.: Model-checking in dense real-time. Inf. Comput. **104**(1), 2–34 (1993)
10. Alur, R., Courcoubetis, C., Henzinger, T.A.: The observational power of clocks. In: Jonsson, B., Parrow, J. (eds.) Intl. Conf. on Concurrency Theory (CONCUR). LNCS, vol. 836, pp. 162–177. Springer, Heidelberg (1994)
11. Alur, R., Courcoubetis, C., Henzinger, T.A.: Computing accumulated delays in real time systems. Form. Methods Syst. Des. **11**(2), 137–155 (1997)
12. Alur, R., Courcoubetis, C., Henzinger, T.A., Ho, P.-H.: Hybrid automata: an algorithmic approach to the specification and verification of hybrid systems. In: Grossman, R.L., Nerode, A., Ravn, A.P., Rischel, H. (eds.) Hybrid Systems (HSCC). LNCS, vol. 736, pp. 209–229. Springer, Heidelberg (1993)
13. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. **126**(2), 183–235 (1994)
14. Alur, R., Feder, T., Henzinger, T.A.: The benefits of relaxing punctuality. J. ACM **43**(1), 116–146 (1996)
15. Alur, R., Fix, L., Henzinger, T.A.: Event-clock automata: a determinizable class of timed automata. In: Dill, D.L. (ed.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 818, pp. 1–13. Springer, Heidelberg (1994)
16. Alur, R., Fix, L., Henzinger, T.A.: Event-clock automata: a determinizable class of timed automata. Theor. Comput. Sci. **211**(1–2), 253–273 (1999)
17. Alur, R., Henzinger, T.A.: Real-time logics: complexity and expressiveness. Inf. Comput. **104**(1), 35–77 (1993)
18. Alur, R., Henzinger, T.A.: A really temporal logic. J. ACM **41**(1), 181–203 (1994)
19. Alur, R., Henzinger, T.A., Vardi, M.Y.: Parametric real-time reasoning. In: Kosaraju, S.R., Johnson, D.S., Aggarwal, A. (eds.) Symposium on the Theory of Computing (STOC), pp. 592–601. ACM, New York (1993)
20. Alur, R., La Torre, S., Pappas, G.J.: Optimal paths in weighted timed automata. In: Domenica Di Benedetto, M., Sangiovani-Vincentelli, A.L. (eds.) Int. Workshop on Hybrid Systems: Computation and Control (HSCC). LNCS, vol. 2034, pp. 49–62. Springer, Heidelberg (2001)
21. Amnell, T., Fersman, E., Mokrushin, L., Pettersson, P., Yi, W.: Times—a tool for modelling and implementation of embedded systems. In: Katoen, J.-P., Stevens, P. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 2280, pp. 460–464. Springer, Heidelberg (2002)
22. Archer, M., Heitmeyer, C.L., Riccobene, E.: Using TAME to prove invariants of automata models: two case studies. In: Per, M., Heimdahl, E. (eds.) Workshop on Formal Methods in Software Practice (FMSP), pp. 25–36. ACM, New York (2000)
23. Arnold, A., Nivat, M.: The metric space of infinite trees. Algebraic and topological properties. Fundam. Inform. **3**(4), 445–476 (1980)
24. Asarin, E., Bozga, M., Kerbrat, A., Maler, O., Pnueli, A., Rasse, A.: Data-structures for the verification of timed automata. In: Maler, O. (ed.) International Workshop on Hybrid and Real-Time Systems (HART). LNCS, vol. 1201, pp. 346–360. Springer, Heidelberg (1997)
25. Asarin, E., Maler, O.: As soon as possible: time optimal control for timed automata. In: Vaandrager, F., van Schuppen, J.H. (eds.) Int. Workshop on Hybrid Systems: Computation and Control (HSCC). LNCS, vol. 1569, pp. 19–30. Springer, Heidelberg (1999)
26. Asarin, E., Maler, O., Pnueli, A.: Symbolic controller synthesis for discrete and timed systems. In: Antsaklis, P., Kohn, W., Nerode, A., Sastry, S. (eds.) Hybrid Systems II (HSCC). LNCS, vol. 999, pp. 1–20. Springer, Heidelberg (1995)

27. Asarin, E., Maler, O., Pnueli, A.: On discretization of delays in timed automata and digital circuits. In: Sangiorgi, D., de Simone, R. (eds.) Intl. Conf. on Concurrency Theory (CONCUR). LNCS, vol. 1466, pp. 470–484. Springer, Heidelberg (1998)
28. Baier, C., Bertrand, N., Bouyer, P., Brihaye, T.: When are timed automata determinizable? In: Albers, S., Alberto, M., Matias, Y., Nikoletseas, S., Thomas, W. (eds.) International Colloquium on Automata, Languages and Programming (ICALP). LNCS, vol. 5557, pp. 43–54. Springer, Heidelberg (2009)
29. Behrmann, G., Bouyer, P., Fleury, E., Larsen, K.G.: Static guard analysis in timed automata verification. In: Garavel, H., Hatcliff, J. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 2619, pp. 254–270. Springer, Heidelberg (2003)
30. Behrmann, G., Bouyer, P., Larsen, K.G., Pelánek, R.: Lower and upper bounds in zone-based abstractions of timed automata. In: Jensen, K., Podelski, A. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 2988, pp. 312–326. Springer, Heidelberg (2004)
31. Behrmann, G., Bouyer, P., Larsen, K.G., Pelánek, R.: Lower and upper bounds in zone-based abstractions of timed automata. Int. J. Softw. Tools Technol. Transf. **8**(3), 204–215 (2006)
32. Behrmann, G., Fehnker, A., Hune, T., Larsen, K.G., Pettersson, P., Romijn, J.: Efficient guiding towards cost-optimality in Uppaal. In: Margaria, T., Yi, W. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 2031, pp. 174–188. Springer, Heidelberg (2001)
33. Behrmann, G., Fehnker, A., Hune, T., Larsen, K.G., Pettersson, P., Romijn, J., Vaandrager, F.: Minimum-cost reachability for priced timed automata. In: Domenica Di Benedetto, M., Sangiovani-Vincentelli, A.L. (eds.) Int. Workshop on Hybrid Systems: Computation and Control (HSCC). LNCS, vol. 2034, pp. 147–161. Springer, Heidelberg (2001)
34. Behrmann, G., Larsen, K.G., Pearson, J., Weise, C., Yi, W.: Efficient timed reachability analysis using clock difference diagrams. In: Halbwachs, N., Peled, D.A. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 1633, pp. 341–353. Springer, Heidelberg (1999)
35. Behrmann, G., Larsen, K.G., Pelánek, R.: To store or not to store. In: Hunt, W.A. Jr, Somenzi, F. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 2725, pp. 433–445. Springer, Heidelberg (2003)
36. Ben Salah, R., Bozga, M., Maler, O.: On interleaving in timed automata. In: Baier, C., Hermanns, H. (eds.) Intl. Conf. on Concurrency Theory (CONCUR). LNCS, vol. 4137, pp. 465–476. Springer, Heidelberg (2006)
37. Bengtsson, J., Jonsson, B., Lilius, J., Yi, W.: Partial order reductions for timed systems. In: Sangiorgi, D., de Simone, R. (eds.) Intl. Conf. on Concurrency Theory (CONCUR). LNCS, vol. 1466, pp. 485–500. Springer, Heidelberg (1998)
38. Bengtsson, J., Yi, W.: On clock difference constraints and termination in reachability analysis of timed automata. In: Dong, J.S., Woodcock, J. (eds.) International Conference on Formal Engineering Methods (ICFEM). LNCS, vol. 2885, pp. 491–503. Springer, Heidelberg (2003)
39. Bengtsson, J., Yi, W.: Timed automata: semantics, algorithms and tools. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) Lectures on Concurrency and Petri Nets. LNCS, vol. 2098, pp. 87–124. Springer, Heidelberg (2004)
40. Bérard, B., Diekert, V., Gastin, P., Petit, A.: Characterization of the expressive power of silent transitions in timed automata. Fundam. Inform. **36**(2–3), 145–182 (1998)
41. Bérard, B., Dufour, C.: Timed automata and additive clock constraints. Inf. Process. Lett. **75**(1–2), 1–7 (2000)
42. Bérard, B., Gastin, P., Petit, A.: Timed automata with non-observable actions: expressive power and refinement. In: Puech, C., Reischuk, R. (eds.) Symposium on Theoretical Aspects of Computer Science (STACS). LNCS, vol. 1046, pp. 257–268. Springer, Heidelberg (1996)
43. Berthomieu, B., Menasche, M.: An enumerative approach for analyzing time Petri nets. In: Mason, R.E.A. (ed.) IFIP World Computer Congress (WCC), pp. 41–46. North-Holland/IFIP, Amsterdam (1983)

44. Beyer, D., Lewerentz, C., Noack, A.: Rabbit: a tool for BDD-based verification of real-time systems. In: Hunt, W.A. Jr, Somenzi, F. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 2725, pp. 122–125. Springer, Heidelberg (2003)
45. Bøgholm, T., Kragh-Hansen, H., Olsen, P., Thomsen, B., Larsen, K.G.: Model-based schedulability analysis of safety critical hard real-time Java programs. In: Bollella, G., Locke, C.D. (eds.) International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES), vol. 343, pp. 106–114. ACM, New York (2008)
46. Bornot, S., Sifakis, J., Tripakis, S.: Modeling urgency in timed systems. In: de Roever, W.-P., Langmaack, H., Pnueli, A. (eds.) International Symposium on Compositionality: The Significant Difference (COMPOS). LNCS, vol. 1536, pp. 103–129. Springer, Heidelberg (1998)
47. Bouajjani, A., Tripakis, S., Yovine, S.: On-the-fly symbolic model checking for real-time systems. In: Symposium on Real-Time Systems (RTSS), pp. 25–35. IEEE, Piscataway (1997)
48. Bouyer, P.: Untameable timed automata! In: Alt, H., Habib, M. (eds.) Symposium on Theoretical Aspects of Computer Science (STACS). LNCS, vol. 2607, pp. 620–631. Springer, Heidelberg (2003)
49. Bouyer, P., Brihaye, T., Markey, N.: Improved undecidability results on weighted timed automata. Inf. Process. Lett. **98**(5), 188–194 (2006)
50. Bouyer, P., Brinksma, E., Larsen, K.G.: Staying alive as cheaply as possible. Form. Methods Syst. Des. **32**(1), 2–23 (2008)
51. Bouyer, P., Cassez, F., Fleury, E., Larsen, K.G.: Optimal strategies in priced timed game automata. In: Kamal, L., Mahajan, M. (eds.) Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS). LNCS, vol. 3328, pp. 148–160. Springer, Heidelberg (2004)
52. Bouyer, P., Chevalier, F., Markey, N.: On the expressiveness of TPTL and MTL. Inf. Comput. **208**(2), 97–116 (2010)
53. Bouyer, P., Dufour, C., Fleury, E., Petit, A.: Updatable timed automata. Theor. Comput. Sci. **321**(2–3), 291–345 (2004)
54. Bouyer, P., Fahrenberg, U., Larsen, K.G., Markey, N.: Timed automata with observers under energy constraints. In: Johansson, K.H., Yi, W. (eds.) Int. Workshop on Hybrid Systems: Computation and Control (HSCC), pp. 61–70. ACM, New York (2010)
55. Bouyer, P., Fahrenberg, U., Larsen, K.G., Markey, N., Srba, J.: Infinite runs in weighted timed automata with energy constraints. In: Cassez, F., Jard, C. (eds.) International Conference on Formal Modelling and Analysis of Timed Systems (FORMATS). LNCS, vol. 5215, pp. 33–47. Springer, Heidelberg (2008)
56. Bouyer, P., Haddad, S., Reynier, P.-A.: Undecidability results for timed automata with silent transitions. Fundam. Inform. **92**(1–2), 1–25 (2009)
57. Bouyer, P., Larsen, K.G., Markey, N.: Model checking one-clock priced timed automata. Log. Methods Comput. Sci. **4**(2), 1–28 (2008)
58. Bouyer, P., Larsen, K.G., Markey, N.: Lower-bound constrained runs in weighted timed automata. In: Int. Conf. on Quantitative Evaluation of Systems (QEST), pp. 128–137. IEEE, Piscataway (2012)
59. Bouyer, P., Larsen, K.G., Markey, N., Rasmussen, J.I.: Almost optimal strategies in one-clock priced timed automata. In: Arun-Kumar, S., Garg, N. (eds.) Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS). LNCS, vol. 4337, pp. 345–356. Springer, Heidelberg (2006)
60. Bozga, M., Jianmin, H., Maler, O., Yovine, S.: Verification of asynchronous circuits using timed automata. In: Asarin, E., Maler, O., Yovine, S. (eds.) Workshop on Theory and Practice of Timed Systems (TPTS). Electronic Notes in Theoretical Computer Science, vol. 65, pp. 47–59. Elsevier, Amsterdam (2002)
61. Brekling, A.W., Hansen, M.R., Madsen, J.: Models and formal verification of multiprocessor system-on-chips. J. Log. Algebraic Program. **77**(1–2), 1–19 (2008)
62. Brenguier, R.: Nash equilibria in concurrent games—application to timed games. PhD thesis, Lab. Spécification & Vérification, ENS Cachan, France (2012)

63. Brihaye, T., Bruyère, V., Raskin, J.-F.: Model-checking for weighted timed automata. In: Lakhnech, Y., Yovine, S. (eds.) Joint Int. Conf. on Formal Modelling and Analysis of Timed Systems (FORMATS) and Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT). LNCS, vol. 3253, pp. 277–292. Springer, Heidelberg (2004)
64. Brihaye, T., Bruyère, V., Raskin, J.-F.: On optimal timed strategies. In: Pettersson, P., Yi, W. (eds.) International Conference on Formal Modelling and Analysis of Timed Systems (FORMATS). LNCS, vol. 3829, pp. 49–64. Springer, Heidelberg (2005)
65. Brihaye, T., Henzinger, T.A., Prabhu, V.S., Raskin, J.-F.: Minimum-time reachability in timed games. In: Arge, L., Cachin, C., Jurdziński, T., Tarlecki, A. (eds.) International Colloquium on Automata, Languages and Programming (ICALP). LNCS, vol. 4596, pp. 825–837. Springer, Heidelberg (2007)
66. Brihaye, T., Laroussinie, F., Markey, N., Oreiby, G.: Timed concurrent game structures. In: Caires, L., Vasconcelos, V.T. (eds.) Intl. Conf. on Concurrency Theory (CONCUR). LNCS, vol. 4703, pp. 445–459. Springer, Heidelberg (2007)
67. Brzozowski, J.A., Seger, C.-J.H.: Advances in asynchronous circuit theory part II: bounded inertial delay models, MOS circuits, design techniques. Bull. Eur. Assoc. Theor. Comput. Sci. **43**, 199–263 (1991)
68. Byg, J., Jørgensen, K.Y., Srba, J.: TAPAAL: editor, simulator and verifier of timed-arc Petri nets. In: Liu, Z., Ravn, A.P. (eds.) Intl. Symp. Automated Technology for Verification and Analysis (ATVA). LNCS, vol. 5799, pp. 84–89. Springer, Heidelberg (2009)
69. Cassez, F., David, A., Fleury, E., Larsen, K.G., Lime, D.: Efficient on-the-fly algorithms for the analysis of timed games. In: Abadi, M., de Alfaro, L. (eds.) Intl. Conf. on Concurrency Theory (CONCUR). LNCS, vol. 3653, pp. 66–80. Springer, Heidelberg (2005)
70. Cassez, F., David, A., Larsen, K.G., Lime, D., Raskin, J.-F.: Timed control with observation based and stuttering invariant strategies. In: Namjoshi, K., Yoneda, T., Higashino, T., Okamura, Y. (eds.) Intl. Symp. Automated Technology for Verification and Analysis (ATVA). LNCS, vol. 4762, pp. 192–206. Springer, Heidelberg (2007)
71. Cassez, F., Jensen, J.J., Larsen, K.G., Raskin, J.-F., Reynier, P.-A.: Automatic synthesis of robust and optimal controllers—an industrial case study. In: Majumdar, R., Tabuada, P. (eds.) Int. Workshop on Hybrid Systems: Computation and Control (HSCC). LNCS, vol. 5469, pp. 90–104. Springer, Heidelberg (2009)
72. Cassez, F., Larsen, K.G.: The impressive power of stopwatches. In: Palamidessi, C. (ed.) Intl. Conf. on Concurrency Theory (CONCUR). LNCS, vol. 1877, pp. 138–152. Springer, Heidelberg (2000)
73. Čerāns, K.: Decidability of bisimulation equivalences for parallel timer processes. In: von Bochmann, G., Probst, D.K. (eds.) Intl. Workshop on Computer-Aided Verification (CAV). LNCS, vol. 663, pp. 302–315. Springer, Heidelberg (1993)
74. Chakrabarti, A., de Alfaro, L., Henzinger, T.A., Stoelinga, M.: Resource interfaces. In: Alur, R., Lee, I. (eds.) Int. Conf. on Embedded Software (EMSOFT). LNCS, vol. 2855, pp. 117–133. Springer, Heidelberg (2003)
75. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Trans. Program. Lang. Syst. **8**(2), 244–263 (1986)
76. Cotton, S., Asarin, E., Maler, O., Niebert, P.: Some progress in satisfiability checking for difference logic. In: Lakhnech, Y., Yovine, S. (eds.) Joint Int. Conf. on Formal Modelling and Analysis of Timed Systems (FORMATS) and Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT). LNCS, vol. 3253, pp. 263–276. Springer, Heidelberg (2004)
77. Cotton, S., Maler, O.: Fast and flexible difference constraint propagation for DPLL(T). In: Biere, A., Gomes, C.P. (eds.) International Conference on Theory and Applications of Satisfiability Testing (SAT). LNCS, vol. 4121, pp. 170–183. Springer, Heidelberg (2006)
78. David, A., Larsen, K.G., Legay, A., Mikučionis, M.: Schedulability of Herschel-Planck revisited using statistical model checking. In: Margaria, T., Steffen, B. (eds.) International Symposium on Leveraging Applications of Formal Methods (ISoLA). LNCS, vol. 7610, pp. 293–307. Springer, Heidelberg (2012)

79. David, A., Larsen, K.G., Legay, A., Mikučionis, M., Wang, Z.: Time for statistical model checking of real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 6806, pp. 349–355. Springer, Heidelberg (2011)
80. David, A., Larsen, K.G., Rasmussen, J.I., Skou, A.: Model-based design for embedded systems. In: Nicolescu, G., Mosterman, P.J. (eds.) Model-Based Design for Embedded Systems. Computational Analysis, Synthesis, and Design of Dynamic Systems. CRC Press, Boca Raton (2009)
81. Daws, C., Tripakis, S.: Model checking of real-time reachability properties using abstractions. In: Steffen, B. (ed.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 1384, pp. 313–329. Springer, Heidelberg (1998)
82. de Alfaro, L., Faella, M., Henzinger, T.A., Majumdar, R., Stoelinga, M.: The element of surprise in timed games. In: Amadio, R., Lugiez, D. (eds.) Intl. Conf. on Concurrency Theory (CONCUR). LNCS, vol. 2761, pp. 142–156. Springer, Heidelberg (2003)
83. De Wulf, M., Doyen, L., Markey, N., Raskin, J.-F.: Robust safety of timed automata. Form. Methods Syst. Des. **33**(1–3), 45–84 (2008)
84. Dembinski, P., Janowska, A., Janowski, P., Penczek, W., Pólrola, A., Szreter, M., Woźna, B., Zbrzezny, A.: Verics: a tool for verifying timed automata and Estelle specifications. In: Garavel, H., Hatcliff, J. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 2619, pp. 278–283. Springer, Heidelberg (2003)
85. Dierks, H.: PLC-automata: a new class of implementable real-time automata. Theor. Comput. Sci. **253**(1), 61–93 (2001)
86. Dierks, H.: Finding optimal plans for domains with continuous effects with UPPAAL Cora. In: ICAPS Workshop on Verification and Validation of Model-Based Planning and Schedulin Systems (VV&PS) (2005)
87. Dill, D.L.: Timing assumptions and verification of finite-state concurrent systems. In: Sifakis, J. (ed.) International Workshop on Automatic Verification Methods for Finite State Systems (AVMFSS). LNCS, vol. 407, pp. 197–212. Springer, Heidelberg (1990)
88. Ehlers, R., Fass, D., Gerke, M., Peter, H.-J.: Fully symbolic timed model checking using constraint matrix diagrams. In: Symposium on Real-Time Systems (RTSS), pp. 360–371. IEEE, Piscataway (2010)
89. Faella, M., La Torre, S., Murano, A.: Dense real-time games. In: Symp. on Logic in Computer Science (LICS), pp. 167–176. IEEE, Piscataway (2002)
90. Fersman, E., Mokrushin, L., Pettersson, P., Yi, W.: Schedulability analysis of fixed-priority systems using timed automata. Theor. Comput. Sci. **354**(2), 301–317 (2006)
91. Gardey, G., Lime, D., Magnin, M., Roux, O.H.: Romeo: a tool for analyzing time Petri nets. In: Etessami, K., Rajamani, S. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 3576, pp. 418–423. Springer, Heidelberg (2005)
92. Gupta, V., Henzinger, T.A., Jagadeesan, R.: Robust timed automata. In: Maler, O. (ed.) International Workshop on Hybrid and Real-Time Systems (HART). LNCS, vol. 1201, pp. 331–345. Springer, Heidelberg (1997)
93. Hansen, M.R., Madsen, J., Brekling, A.W.: Semantics and verification of a language for modelling hardware architectures. In: Jones, C.B., Liu, Z., Woodcock, J. (eds.) Formal Methods and Hybrid Real-Time Systems, Essays in Honor of Dines Bjørner and Zhou Chaochen on the Occasion of Their 70th Birthdays. LNCS, vol. 4700, pp. 300–319. Springer, Heidelberg (2007)
94. Hansen, T.D., Ibsen-Jensen, R., Miltersen, P.B.: A faster algorithm for solving one-clock priced timed games. In: D'Argenio, P.R., Melgratt, H.C. (eds.) Intl. Conf. on Concurrency Theory (CONCUR). LNCS, vol. 8052, pp. 531–545. Springer, Heidelberg (2013)
95. Havelund, K., Skou, A., Larsen, K.G., Lund, K.: Formal modelling and analysis of an audio/video protocol: an industrial case study using Uppaal. In: Symposium on Real-Time Systems (RTSS), pp. 2–13. IEEE, Piscataway (1997)

96. Hendriks, M., Behrmann, G., Larsen, K.G., Niebert, P., Vaandrager, F.: Adding symmetry reduction to Uppaal. In: Larsen, K.G., Niebert, P. (eds.) Int. Workshop on Formal Modelling and Analysis of Timed Systems (FORMATS). Lecture Notes in Computer Science, vol. 2791. Springer, Heidelberg (2003)
97. Hendriks, M., van den Nieuwelaar, B., Vaandrager, F.: Model checker aided design of a controller for a wafer scanner. Int. J. Softw. Tools Technol. Transf. **8**(6), 633–647 (2006)
98. Henzinger, T.A.: The theory of hybrid automata. In: Symp. on Logic in Computer Science (LICS), pp. 278–292. IEEE, Piscataway (1996)
99. Henzinger, T.A., Ho, P.-H., Wong-Toi, H.: A user guide to HyTech. In: Brinksma, E., Cleaveland, R., Larsen, K.G., Margaria, T., Steffen, B. (eds.) Intl. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 1019, pp. 41–71. Springer, Heidelberg (1995)
100. Henzinger, T.A., Kopke, P.W., Puri, A., Varaiya, P.: What is decidable about hybrid automata? J. Comput. Syst. Sci. **57**(1), 94–124 (1998)
101. Henzinger, T.A., Kopke, P.W., Wong-Toi, H.: The expressive power of clocks. In: Fülöp, Z., Gecseg, F. (eds.) International Colloquium on Automata, Languages and Programming (ICALP). LNCS, vol. 944, pp. 417–428. Springer, Heidelberg (1995)
102. Henzinger, T.A., Manna, Z., Pnueli, A.: What good are digital clocks? In: Kuich, W. (ed.) International Colloquium on Automata, Languages and Programming (ICALP). LNCS, vol. 623, pp. 545–558. Springer, Heidelberg (1992)
103. Henzinger, T.A., Nicollin, X., Sifakis, J., Yovine, S.: Symbolic model checking for real time systems. Inf. Comput. **111**(2), 193–244 (1994)
104. Henzinger, T.A., Raskin, J.-F., Schobbens, P.-Y.: The regular real-time languages. In: Larsen, K.G., Skyum, S., Winskel, G. (eds.) International Colloquium on Automata, Languages and Programming (ICALP). LNCS, vol. 1443, pp. 580–591. Springer, Heidelberg (1998)
105. Herrmann, P.: Timed automata and recognizability. Inf. Process. Lett. **65**(6), 313–318 (1998)
106. Hune, T., Romijn, J., Stoelinga, M., Vaandrager, F.: Linear parametric model checking of timed automata. J. Log. Algebraic Program. **52–53**, 183–220 (2002)
107. Hüttel, H., Larsen, K.G.: The use of static constructs in a modal process logic. In: Meyer, A.R., Taitslin, M.A. (eds.) Symposium on Logical Foundations of Computer Science. LNCS, vol. 363, pp. 163–180. Springer, Heidelberg (1989)
108. Igna, G., Kannan, V., Yang, Y., Basten, T., Geilen, M.C.W., Vaandrager, F., Voorhoeve, M., de Smet, S., Somers, L.J.: Formal modeling and scheduling of datapaths of digital document printers. In: Cassez, F., Jard, C. (eds.) International Conference on Formal Modelling and Analysis of Timed Systems (FORMATS). LNCS, vol. 5215, pp. 170–187. Springer, Heidelberg (2008)
109. Jaghoori, M.M., de Boer, F.S., Chothia, T., Sirjani, M.: Schedulability of asynchronous real-time concurrent objects. J. Log. Algebraic Program. **78**(5), 402–416 (2009)
110. Jensen, J.J., Rasmussen, J.I., Larsen, K.G., David, A.: Guided controller synthesis for climate controller using UPPAAL Tiga. In: Raskin, J.-F., Thiagarajan, P.S. (eds.) International Conference on Formal Modelling and Analysis of Timed Systems (FORMATS). LNCS, vol. 4763, pp. 227–240. Springer, Heidelberg (2007)
111. Jørgensen, K.Y., Larsen, K.G., Srba, J.: Time-darts: a data structure for verification of closed timed automata. In: Conference on Systems Software Verification (SSV). Electronic Proceedings in Theoretical Computer Science, vol. 102, pp. 141–155 (2012)
112. Julliand, J., Mountassir, H., Oudot, É.: VeSTA: a tool to verify the correct integration of a component in a composite timed system. In: Butler, M.J., Hinchey, M.G., Larrondo-Petrie, M.M. (eds.) International Conference on Formal Engineering Methods (ICFEM). LNCS, vol. 4789, pp. 116–135. Springer, Heidelberg (2007)
113. Klimoš, M., Larsen, K.G., Štefaňák, F., Thaarup, J.: Nash equilibria in concurrent priced games. In: Dediu, A.H., Martín-Vide, C. (eds.) Intl. Conf. on Language and Automata Theory and Applications (LATA). LNCS, vol. 7183, pp. 363–376. Springer, Heidelberg (2012)

114. Kupferschmid, S., Dräger, K., Hoffmann, J., Finkbeiner, B., Dierks, H., Podelski, A., Behrmann, G.: Uppaal/DMC—abstraction-based heuristics for directed model checking. In: Grumberg, O., Huth, M. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 4424, pp. 679–682. Springer, Heidelberg (2007)
115. Kupferschmid, S., Hoffmann, J., Larsen, K.G.: Fast directed model checking via Russian doll abstraction. In: Ramakrishnan, C.R., Rehof, J. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 4963, pp. 203–217. Springer, Heidelberg (2008)
116. Kupferschmid, S., Wehrle, M., Nebel, B., Podelski, A.: Faster than Uppaal? In: Gupta, A., Malik, S. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 5123, pp. 552–555. Springer, Heidelberg (2008)
117. La Torre, S., Napoli, M.: A decidable dense branching-time temporal logic. In: Kapoor, S., Prasad, S. (eds.) Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS). LNCS, vol. 1974, pp. 139–150. Springer, Heidelberg (2000)
118. La Torre, S., Napoli, M.: Timed tree automata with an application to temporal logic. Acta Inform. **38**(2), 89–116 (2001)
119. Larsen, K.G., Behrmann, G., Brinksma, E., Fehnker, A., Hune, T., Pettersson, P., Romijn, J.: As cheap as possible: efficient cost-optimal reachability for priced timed automata. In: Berry, G., Comon, H., Finkel, A. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 2102, pp. 493–505. Springer, Heidelberg (2001)
120. Larsen, K.G., Larsson, F., Pettersson, P., Yi, W.: Efficient verification of real-time systems: compact data structure and state-space reduction. In: Symposium on Real-Time Systems (RTSS), pp. 14–24. IEEE, Piscataway (1997)
121. Larsen, K.G., Pearson, J., Weise, C., Yi, W.: Clock difference diagrams. Nord. J. Comput. **6**(3), 271–298 (1999)
122. Larsen, K.G., Pettersson, P., Yi, W.: Uppaal in a nutshell. Int. J. Softw. Tools Technol. Transf. **1**(1–2), 134–152 (1997)
123. Larsen, K.G., Rasmussen, J.I.: Optimal conditional reachability for multi-priced timed automata. Theor. Comput. Sci. **390**(2–3), 197–213 (2008)
124. Lasota, S., Walukiewicz, I.: Alternating timed automata. ACM Trans. Comput. Log. **9**(2), 10:1–10:27 (2008)
125. Lugiez, D., Niebert, P., Zennou, S.: A partial order semantics approach to the clock explosion problem of timed automata. In: Jensen, K., Podelski, A. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 2988, pp. 296–311. Springer, Heidelberg (2004)
126. Lugiez, D., Niebert, P., Zennou, S.: A partial order semantics approach to the clock explosion problem of timed automata. Theor. Comput. Sci. **345**(1), 27–59 (2005)
127. Mader, A., Wupper, H.: Timed automaton models for simple programmable logic controllers. In: Euromicro Conference on Real-Time Systems (ECRTS), pp. 106–113. IEEE, Piscataway (1999)
128. Maler, O., Pnueli, A., Sifakis, J.: On the synthesis of discrete controllers for timed systems (an extended abstract). In: Mayr, E.W., Puech, C. (eds.) Symposium on Theoretical Aspects of Computer Science (STACS). LNCS, vol. 900, pp. 229–242. Springer, Heidelberg (1995)
129. Miller, J.S.: Decidability and complexity results for timed automata and semi-linear hybrid automata. In: Lynch, N., Krogh, B.H. (eds.) Int. Workshop on Hybrid Systems: Computation and Control (HSCC). LNCS, vol. 1790, pp. 296–310. Springer, Heidelberg (2000)
130. Milner, R.: Communication and Concurrency. Prentice Hall, Upper Saddle River (1989)
131. Minea, M.: Partial order reduction for model checking of timed automata. In: Baeten, J.C.M., Mauw, S. (eds.) Intl. Conf. on Concurrency Theory (CONCUR). LNCS, vol. 1664, pp. 431–446. Springer, Heidelberg (1999)
132. Møller, J.B., Lichtenberg, J., Andersen, H.R., Hulgaard, H.: Difference decision diagrams. In: Flum, J., Rodríguez-Artalejo, M. (eds.) International Workshop on Computer Science Logic (CSL). LNCS, vol. 1862, pp. 111–125. Springer, Heidelberg (1999)

133. Ouaknine, J., Rabinovich, A., Worrell, J.: Time-bounded verification. In: Bravetti, M., Zavattaro, G. (eds.) Intl. Conf. on Concurrency Theory (CONCUR). LNCS, vol. 5710, pp. 496–510. Springer, Heidelberg (2009)
134. Ouaknine, J., Worrell, J.: On the language inclusion problem for timed automata: closing a decidability gap. In: Symp. on Logic in Computer Science (LICS), pp. 54–63. IEEE, Piscataway (2004)
135. Ouaknine, J., Worrell, J.: On the decidability of metric temporal logic. In: Symp. on Logic in Computer Science (LICS), pp. 188–197. IEEE, Piscataway (2005)
136. Ouaknine, J., Worrell, J.: On metric temporal logic and faulty Turing machines. In: Aceto, L., Ingólfsdóttir, A. (eds.) International Conference on Foundations of Software Science and Computation Structure (FoSSaCS. LNCS, vol. 3921, pp. 217–230. Springer, Heidelberg (2006)
137. Ouaknine, J., Worrell, J.: On the decidability and complexity of metric temporal logic over finite words. Log. Methods Comput. Sci. **3**(1), 1–27 (2007)
138. Park, D.M.R.: Concurrency and automata on infinite sequences. In: Deussen, P. (ed.) GI-Conference on Theoretical Computer Science (TCS). LNCS, vol. 104, pp. 167–183. Springer, Heidelberg (1981)
139. Puri, A.: Dynamical properties of timed automata. In: Ravn, A.P., Rischel, H. (eds.) Int. Symp. on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT). LNCS, vol. 1486, pp. 210–227. Springer, Heidelberg (1998)
140. Raskin, J.-F., Schobbens, P.-Y.: The logic of event-clocks: decidability, complexity and expressiveness. J. Autom. Lang. Comb. **4**(3), 247–286 (1999)
141. Roux, O.F., Rusu, V.: Deciding time-bounded properties for ELECTRE reactive programs with stopwatch automata. In: Antsaklis, P., Kohn, W., Nerode, A., Sastry, S. (eds.) Hybrid Systems II (HSCC). LNCS, vol. 999, pp. 405–416. Springer, Heidelberg (1995)
142. Rutkowski, M.: Two-player reachability-price games on single-clock timed automata. In: Workshop on Quantitative Aspects of Programming Languages (QAPL). Electronic Proceedings in Theoretical Computer Science, vol. 57 (2011)
143. Sankur, O.: Robustness in timed automata: analysis, synthesis, implementation. PhD thesis, Lab. Spécification & Vérification, ENS Cachan, France (2013)
144. Schuts, M., Zhu, Y., Heidarian, F., Vaandrager, F.: Modelling clock synchronization in the Chess gMAC WSN protocol. In: Andova, S., McIver, A.K., D'Argenio, P.R., Cuijpers, P.J.L., Markovski, J., Morgan, C.C., Núñez, M. (eds.) Workshop on Quantitative Formal Methods: Theory and Applications (QFM). Electronic Proceedings in Theoretical Computer Science, vol. 13, pp. 41–54 (2009)
145. Tapken, J., Dierks, H.: MOBY/PLC—graphical development of PLC-automata. In: Ravn, A.P., Rischel, H. (eds.) Int. Symp. on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT). LNCS, vol. 1486, pp. 311–314. Springer, Heidelberg (1998)
146. Tripakis, S.: Description and schedulability analysis of the software architecture of an automated vehicle control system. In: Sangiovani-Vincentelli, A.L., Sifakis, J. (eds.) Int. Conf. on Embedded Software (EMSOFT). LNCS, vol. 2491, pp. 123–137. Springer, Heidelberg (2002)
147. Tripakis, S.: Checking timed Büchi automata emptiness on simulation graphs. ACM Trans. Comput. Log. **10**(3), 15:1–15:19 (2009)
148. Tripakis, S., Altisen, K.: On-the-fly controller synthesis for discrete and dense-time systems. In: Wing, J.M., Woodcock, J., Davies, J. (eds.) World Congress on Formal Methods (FM). LNCS, vol. 1708, pp. 233–252. Springer, Heidelberg (1999)
149. Tripakis, S., Yovine, S.: Analysis of timed systems using time-abstracting bisimulations. Form. Methods Syst. Des. **18**(1), 25–68 (2001)
150. Tripakis, S., Yovine, S.: Timing analysis and code generation of vehicle control software using Taxys. In: Havelund, K., Roşu, G. (eds.) International Workshop on Runtime Verification (RV). Electronic Notes in Theoretical Computer Science, vol. 55, pp. 277–286. Elsevier, Amsterdam (2001)

151. Tripakis, S., Yovine, S., Bouajjani, A.: Checking timed Büchi automata emptiness efficiently. Form. Methods Syst. Des. **26**(3), 267–292 (2005)
152. Wang, F.: Model-checking distributed real-time systems with states, events, and multiple fairness assumptions. In: Rattray, C., Maharaj, S., Shankland, C. (eds.) International Conference on Algebraic Methodology and Software Technology (AMAST). LNCS, vol. 3116, pp. 553–567. Springer, Heidelberg (2004)
153. Wang, F.: Efficient model-checking of dense-time systems with time-convexity analysis. Theor. Comput. Sci. **467**, 89–108 (2013)
154. Wang, F., Huang, G.-D., Yu, F.: TCTL inevitability analysis of dense-time systems: from theory to engineering. IEEE Trans. Softw. Eng. **32**(7), 510–526 (2006)
155. Wang, F., Yao, L.-W., Yang, Y.-L.: Efficient verification of distributed real-time systems with broadcasting behaviors. Real-Time Syst. **47**(4), 285–318 (2011)
156. Waszniowski, L., Hanzálek, Z.: Formal verification of multitasking applications based on timed automata model. Real-Time Syst. **38**(1), 39–65 (2008)
157. Yi, W., Pettersson, P., Daniels, M.: Automatic verification of real-time communicating systems by constraint-solving. In: Hogrefe, D., Leue, S. (eds.) Intl. Conf. on Formal Description Techniques (FORTE). IFIP Conference Proceedings, vol. 6, pp. 243–258. Chapman & Hall, London (1995)
158. Yovine, S.: Kronos: a verification tool for real-time systems. Int. J. Softw. Tools Technol. Transf. **1**(1–2), 123–133 (1997)
159. Zennou, S., Yguel, M., Niebert, Peter: ELSE: a new symbolic state generator for timed automata. In: Larsen, K.G., Niebert, P. (eds.) Int. Workshop on Formal Modelling and Analysis of Timed Systems (FORMATS). LNCS, vol. 2791, pp. 273–280. Springer, Heidelberg (2003)

Chapter 30
Verification of Hybrid Systems

Laurent Doyen, Goran Frehse, George J. Pappas, and André Platzer

Abstract Hybrid systems are models which combine discrete and continuous behavior. They occur frequently in safety-critical applications in various domains such as health care, transportation, and robotics, as a result of interactions between a digital controller and a physical environment. They also have relevance in other areas such as systems biology, in which the discrete dynamics arises as an abstraction of fast continuous processes. One of the prominent models is that of *hybrid automata*, where differential equations are associated with each node, and jump constraints such as guards and resets are associated with each edge.

In this chapter, we focus on the problem of model checking of hybrid automata against reachability and invariance properties, enabling the verification of general temporal logic specifications. We review the main decidability results for hybrid automata, and since model checking is in general undecidable, we present three complementary analysis approaches based on symbolic representations, abstraction, and logic. In particular, we illustrate polyhedron-based reachability analysis, finite quotients, abstraction refinement techniques, and logic-based verification. We survey important tools and application domains of successful hybrid system verification in this vibrant area of research.

L. Doyen
LSV, CNRS & ENS Paris-Saclay, Cachan, France

G. Frehse
Verimag, University Grenoble Alpes, Grenoble, France

G.J. Pappas
University of Pennsylvania, Philadelphia, PA, USA

A. Platzer (✉)
Carnegie Mellon University, Pittsburgh, PA, USA
e-mail: aplatzer@cs.cmu.edu

E.M. Clarke et al. (eds.), *Handbook of Model Checking*,
DOI 10.1007/978-3-319-10575-8_30

30.1 Introduction

Information technology (IT) has dramatically changed our lives. The first revolution in information technology led to the birth of the computer. The second information technology revolution led to the creation of the Internet, connecting computing around the world and resulting in the hyper-connected world that we live in. The third revolution that is now taking place is connecting all the computational power to the physical world. Computing powerhouses, such as Intel, are investing in wearable computing and smart watches, Google has invested in self-driving cars and bought Nest, the makers of a learning thermostat, thereby connecting Google to building control and energy markets, and Amazon is investing in robotics and unmanned aerial vehicles. Similarly the most significant innovation in automotive companies recently came from software-intensive car technology, leading from adaptive cruise control to driverless cars. There is a plethora of novel medical devices, either wearable or implantable, that sense patient vitals and use computer algorithms to diagnose medical conditions or even perform life-critical functions. The fundamental aspect of this third revolution is the fusion of IT with physical devices that interact with the physical world.

The marriage of IT with the physical world is known as embedded computing as it consists of computing that is embedded and tightly interacts with the physical world. A major modeling challenge is how to formally capture the interaction between computing and physics so that we can *reason about the effect of physics on computing and vice versa*. This led to the development of *hybrid systems* [8, 34, 35, 91, 132], where both discrete and continuous behaviors of the system are important. Hybrid systems grew out of the necessity to enrich purely digital models of computing with analog models of physics. As a result, hybrid systems contain both digital models of computing (such as automata or programs) as well as analog elements (such as differential equations) integrated in such a way that one can model many embedded computing applications.

The need for formal models of hybrid systems arises from the fact that many embedded computing problems are safety-critical. They arise in collision avoidance protocols in air traffic control [31, 100, 113, 129, 149, 174–177], cruising controllers for automotive vehicles [24, 46, 53, 62, 101, 114, 164, 168], obstacle avoidance algorithms for autonomous ground robots [130, 182], and software-controlled medical devices that actively regulate life-critical functions or help surgeons with surgical robotic systems [104]. Therefore there is not only a need for formal models of embedded computing, but also for rigorous verification approaches to guarantee that the embedded computation, as modeled by a hybrid system, is formally safe. This has resulted in the development of a new paradigm within the formal methods community, namely the formal verification of hybrid models of embedded computing.

There is a range of formal models for hybrid systems [8, 27, 29, 35, 52, 91, 118, 120, 131, 132, 134, 135, 143, 167, 179, 180], each with different advantages for different purposes. This chapter focuses on *hybrid automata* [8, 91], because they directly generalize the timed automata that have been considered in Chap. 29. The basic idea in hybrid automata is to associate differential equations with the nodes of

an automaton. The automaton structure defines how and under which conditions the system switches between the various differential equations and what happens to the state if they switch. Timed automata, which are discussed in Chap. 29, are a special case of hybrid automata, where all differential equations are of the form $\dot{x} = 1$ such that x is a clock variable measuring the progress of time and additional linearity assumptions are met for the switching conditions. Timed automata are an interesting subclass of hybrid automata, because reachability is decidable in this subclass. For systems with more general continuous dynamics, e.g., moving, acceleration, or curving, however, timed automata are not sufficient, and hybrid systems models are needed instead.

In this chapter, we give a survey of model-checking techniques for hybrid systems with an emphasis on the handling of continuous dynamics. It has been proved that continuous dynamics verification is the most fundamental question in hybrid systems verification [135, 141], because discrete dynamics can be verified exactly as well as continuous dynamics.

In this section, we survey a set of complementary verification techniques for hybrid systems, including explicit-state reachability computations with termination criteria like bounded-horizon (Sect. 30.4, which is related to Chap. 5), abstraction techniques and abstraction refinement loops (Sect. 30.5, also see Chap. 13), and logic-based verification approaches (Sect. 30.6, which is related to Chaps. 2, 26, 20, and 15). Other surveys of several aspects of hybrid systems can be found in the literature [7, 12, 39, 80, 103, 143, 163, 171, 172]. A control-theoretic view on hybrid systems verification has been reported in a book by Tabuada [165]. A logic and proofs view on hybrid systems verification can be found in a book by one of the authors [137]. Introductions to embedded systems from a cyber-physical systems perspective have been reported in the literature [111, 122] and in university courses.

Hybrid systems has become a very active and successful area of research with a vibrant community. Giving a complete overview of all relevant approaches is impossible in this chapter, instead we focus on giving an overview of some of the most important representative classes of techniques. By their very nature, hybrid systems tend to be mathematically demanding, but they can also be exceedingly beautiful. The broad applicability and scope of the resulting hybrid systems analysis techniques make hybrid systems a very rewarding area of science with the potential for significant impact on practical applications.

30.2 Basic Definitions

Hybrid systems combine discrete evolutions (namely, mode changes and variable updates) and continuous evolutions through variables whose dynamics is governed by differential equations. Hybrid system models have been introduced to deal with such systems in a uniform way [8, 27, 29, 35, 52, 91, 118, 120, 131, 132, 134–136, 167, 179, 180]. The original definitions are very general. In this chapter, we focus on subclasses of particular interest. Timed automata are an important class of hybrid automata for which safety verification is decidable (see Chap. 29). When

continuous variables are subject to rectangular flow constraints, that is constraints of the form $\dot{x} \in [a, b]$, hybrid automata are called *rectangular*. For that subclass of hybrid automata, there exists a reasonably efficient algorithm to compute the image of a (simple) set. Based on this algorithm, there exists an iterative method that computes the exact set of reachable states when it terminates. This semi-algorithm can be used to establish or refute *safety properties*. On the other hand, if the evolution of the continuous variables is subject to more complicated flow constraints, for example affine dynamics like $\dot{x} = 3x - y$, computing the flow successor is much more difficult and only approximate methods are known.

30.2.1 Predicates

Let $X = \{x_1, \ldots, x_n\}$ be a finite set of variables. Given a valuation $v : X \to \mathbb{R}$ and $Y \subseteq X$, define $v_{|Y} : Y \to \mathbb{R}$ by $v_{|Y}(x) = v(x)$ for every $x \in Y$.

Definition 1 (Polynomial term) A *polynomial term* over a finite set of variables $X = \{x_1, \ldots, x_n\}$ is an expression of the form $y \equiv \sum_{i \in \mathbb{N}^n} a_i x_1^{i_1} \ldots x_n^{i_n}$ where $a_i \in \mathbb{Q}$ $(i = (i_1, \ldots, i_n) \in \mathbb{N}^n)$ are rational constants and almost all a_i are zero. Given a valuation v over X, we write $[\![y]\!]_v$ for the real number $\sum_{i \in \mathbb{N}^n} a_i v(x_1)^{i_1} \ldots v(x_n)^{i_n}$ obtained by evaluating the polynomial term at v. We denote by $\mathsf{PTerm}(X)$ the set of all polynomial terms over the variables X.

Definition 2 (Polynomial constraint) A *polynomial constraint* over X (also known as a semi-algebraic constraint) is a finite formula φ defined by the following grammar rule:

$$\varphi ::= \theta \bowtie 0 \mid \varphi \wedge \varphi \mid \varphi \vee \varphi$$

where $\theta \in \mathsf{PTerm}(X)$ and $\bowtie \in \{<, \leq, =, >, \geq\}$. We denote by $\mathsf{PConstr}(X)$ the class of polynomial constraints over the set of variables X.

Definition 3 Given a valuation $v : X \to \mathbb{R}$ and a polynomial constraint $\varphi \in \mathsf{PConstr}(X)$, we write $v \models \varphi$ and say that v *satisfies* φ, which we define inductively as:

- $v \models \theta \bowtie 0$ if $[\![\theta]\!]_v \bowtie 0$,
- $v \models \varphi_1 \wedge \varphi_2$ if $v \models \varphi_1$ and $v \models \varphi_2$,
- $v \models \varphi_1 \vee \varphi_2$ if $v \models \varphi_1$ or $v \models \varphi_2$.

We also write $v \in [\![\varphi]\!]$ when $v \models \varphi$. If $v : X \to \mathbb{R}$ and $w : Y \to \mathbb{R}$ are two valuations for disjoint sets of variables X, Y (with $X \cap Y = \emptyset$), we also write $(v, w) \in [\![\varphi]\!]$ when $u \models \varphi$ where $u : X \cup Y \to \mathbb{R}$ is defined such that $u_{|X} = v$ and $u_{|Y} = w$.

Linear constraints are an Important special case of polynomial constraints. The set of solutions of a linear constraint describes a set of polyhedra. This geometric

interpretation is sometimes used for model checking since image computations of polyhedra can be quite efficient.

Definition 4 (Linear constraint) A *linear term* is a polynomial term of the form $y \equiv a_0 + \sum_{x_i \in X} a_i x_i$ with $a_i \in \mathbb{Q}$. We denote the set of all linear terms over X by $\mathsf{LTerm}(X)$. A *linear constraint* is a polynomial constraint where all terms are linear. It is called *conjunctive* if it does not contain any disjunctions. We denote by $\mathsf{LConstr}(X)$ the class of linear constraints over X and by $\mathsf{LConstr}_c(X)$ the class of conjunctivelinear constraints. The constraints true and false are defined as abbreviations in a standard way.

Definition 5 (Polyhedron) A set of valuations that can be defined by a conjunctivelinear constraint is called a *polyhedron*, and a closed and bounded polyhedron is called a *polytope*. We denote a polyhedron in its *constraint representation* as

$$P = \left\{ x \;\middle|\; \bigwedge_{i=0}^{m} a_i^{\mathsf{T}} x \bowtie_i b_i \right\}, \quad \text{with } \bowtie_i \in \{<, \leq, =, >, \geq\},^{1}$$

where the $a_i \in \mathbb{Q}^n$ are called *facet normals* and the $b_i \in \mathbb{Q}^n$ constants. In vector-matrix notation, this corresponds to

$$P = \{x \mid Ax \bowtie b\}, \quad \text{with } A = \begin{pmatrix} a_1^{\mathsf{T}} \\ \vdots \\ a_m^{\mathsf{T}} \end{pmatrix}, \; \bowtie = \begin{pmatrix} \bowtie_1 \\ \vdots \\ \bowtie_m \end{pmatrix}, \; b = \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix}.$$

A closed polyhedron $P \subseteq \mathbb{R}^n$ can be represented by a pair (V, R), called the *generators* of P, where $V \subseteq \mathbb{Q}^n$ is a finite set of *vertices*, and $R \subseteq \mathbb{Q}^n$ is a finite set of *rays*, with:

$$P = \left\{ \sum_{v_i \in V} \lambda_i \cdot v_i + \sum_{r_j \in R} \mu_j \cdot r_j \;\middle|\; \lambda_i \geq 0, \mu_j \geq 0, \sum_i \lambda_i = 1 \right\}.$$

The representation can be extended with *closure points* to deal with non-closed polyhedra [22].

There are algorithms for transforming one representation into the other, namely the Fourier–Motzkin procedure (or quantifier elimination) for computing the system of inequalities from the generators [55, 63], and Chernikova's algorithm for computing the generators from a set of predicates [44].

[1] $x^{\mathsf{T}} y = \sum_{i=1}^{n} x_i y_i$ is the scalar product of n-dimensional vectors $x = (x_1, \ldots, x_n)$ and $y = (y_1, \ldots, y_n)$.

30.2.2 Hybrid Automata

We define hybrid automata with polynomial dynamics [6, 96].

Definition 6 (Hybrid automaton with polynomial dynamics) A *hybrid automaton H with polynomial dynamics* is a tuple

$$\langle \mathsf{Loc}, \mathsf{Lab}, \mathsf{Edg}, X, \mathsf{Init}, \mathsf{Inv}, \mathsf{Flow}, \mathsf{Jump}, \mathsf{Final} \rangle$$

where:

- $\mathsf{Loc} = \{\ell_1, \ldots, \ell_m\}$ is a finite set of *locations*;
- Lab is a finite set of *labels*, including the *silent* label τ;
- $\mathsf{Edg} \subseteq \mathsf{Loc} \times \mathsf{Lab} \times \mathsf{Loc}$ is a finite set of *edges*;
- $X = \{x_1, \ldots, x_n\}$ is a finite set of *variables*;
- $\mathsf{Init} : \mathsf{Loc} \to \mathsf{PConstr}(X)$ gives the *initial condition* $\mathsf{Init}(\ell)$ of location ℓ. The automaton can start in ℓ with an initial valuation v lying in $[\![\mathsf{Init}(\ell)]\!]$;
- $\mathsf{Inv} : \mathsf{Loc} \to \mathsf{PConstr}(X)$ gives the *evolution domain restriction* $\mathsf{Inv}(\ell)$ (also called the invariant) of location ℓ. The automaton can stay in ℓ as long as the values of its variables lie in $[\![\mathsf{Inv}(\ell)]\!]$;
- $\mathsf{Flow} : \mathsf{Loc} \to \mathsf{PConstr}(X \cup \dot{X})$ is the *flow constraint*, which constrains the evolution of the variables in each location. In a location ℓ, if the valuation of the variables is v_0 at time $t = 0$, then at time $t \geq 0$, the value of the variables is $\phi(t)$ where $\phi : \mathbb{R} \to \mathbb{R}^X$ is such that the flow relation $\mathsf{Flow}(\ell)(\phi(t), \dot{\phi}(t))$ holds for the flow $\phi(t)$ and its time-derivative $\dot{\phi}(t)$, and $\phi(0) = v_0$.[2]
- $\mathsf{Jump} : \mathsf{Edg} \to \mathsf{PConstr}(X \cup X^+)$ with $X^+ = \{x_1^+, \ldots, x_n^+\}$ gives the *jump condition* $\mathsf{Jump}(e)$ of edge e. The variables in X^+ refer to the updated values of the variables after the edge has been traversed. Jump conditions are often conjunctions of a guard and a reset constraint. There, the constraints purely on variables in X are called *guards*, and the constraints that describe variables in X^+ in terms of variables in X are called *updates* or *resets*.
- $\mathsf{Final} : \mathsf{Loc} \to \mathsf{PConstr}(X)$ gives the *final condition* $\mathsf{Final}(\ell)$ of location ℓ. Depending on the analysis question at hand, final conditions can either specify the unsafe states of the system or the desired states of the system.

The labels on edges can be used to synchronize hybrid automata in a compositional design. In the rest of this chapter, we assume that a single automaton is to be analyzed.

Example Figure 1 represents an affine automaton modeling a single gas-burner that is shared for heating alternatively two water tanks. It has three locations ℓ_0, ℓ_1, ℓ_2 and two variables x_1 and x_2, the temperature in the two tanks. The gas-burner can

[2] Note that the semantics of flow constraints requires some attention, see differential-algebraic constraints [136].

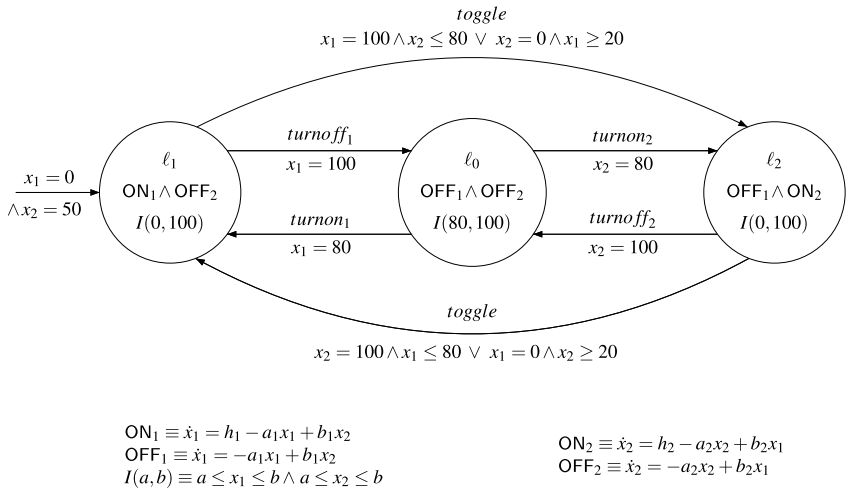
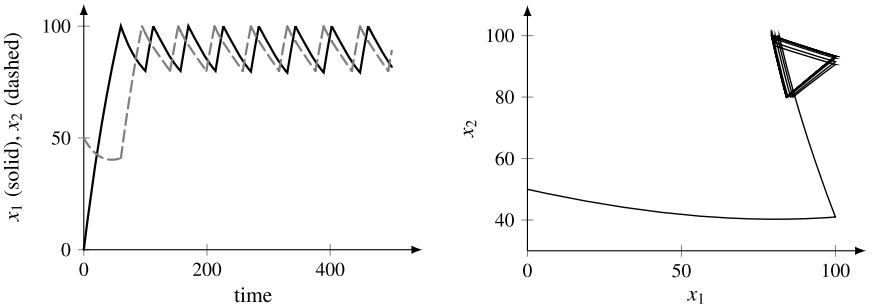


Fig. 1 A shared gas-burner



(a) x_1, x_2 over time (x_1 solid, x_2 dashed) (b) reachable values of variables x_1 and x_2

Fig. 2 The evolution of the continuous variables of the shared gas burner example, starting from location ℓ_1 with initial values $x_1 = 0$ and $x_2 = 50$

be either switched off (in ℓ_0) or turned on heating one of the two tanks (in ℓ_1 or ℓ_2). The dynamics in each location is given by a combination of the predicates ON_i and OFF_i ($i = 1, 2$) where the constants a_i model the heat exchange rate of the tank i with the room in which the tanks are located, b_i model the heat exchange rate between the two tanks and h_i depends on the power of the gas-burner. On every edge of the automaton, we have omitted the condition $x_1^+ = x_1 \wedge x_2^+ = x_2$ also written as $stable(x_1, x_2)$ that asks that the values of the variables are maintained when the edge is traversed. In the sequel, we fix the constants $h_1 = h_2 = 2$, $a_1 = a_2 = 0.01$ and $b_1 = b_2 = 0.005$. The evolution of the continuous variables over time is shown in Fig. 2. Starting in location ℓ_1, the burner heats up tank 1 until it reaches a temperature of 100 degrees. Since the temperature of tank 2 is below 80 degrees, the automaton takes edge *toggle* to location ℓ_2. Note that edge $turnoff_1$ cannot be

taken since the evolution $I(80, 100)$ domain of the target location is not satisfied by x_2. In location ℓ_2, the burner heats up tank 2 until it reaches 100 degrees. Since x_1 is still above 80 degrees, the automaton takes edge $turnoff_2$ to location ℓ_0, where the burner is off. It briefly remains here until x_1 falls to 80 degrees, at which point it takes edge $turnon_1$ to location ℓ_1, where the burner heats up tank 1. The automaton converges towards a limit cycle of heating tank 1, heating tank 2, and briefly turning off the burner.

The definitions above define what a hybrid automaton consists of (flows, jumps, initial regions, ...) but they do not specify the behavior of a hybrid automaton or how its state evolves over time. This is the purpose of defining a semantics for hybrid automata by providing a transition system for each hybrid automaton.

Definition 7 (Semantics of hybrid automata) The *semantics* of a hybrid automaton $H = \langle \mathsf{Loc}, \mathsf{Lab}, \mathsf{Edg}, X, \mathsf{Init}, \mathsf{Inv}, \mathsf{Flow}, \mathsf{Jump}, \mathsf{Final} \rangle$ is the transition system $[\![H]\!] = \langle S, S_0, S_f, \Sigma, \rightarrow \rangle$ where $S = \{(\ell, v) \in \mathsf{Loc} \times \mathbb{R}^X \mid v \in [\![\mathsf{Inv}(\ell)]\!]\}$ is the *state space*, $S_0 = \{(\ell, v) \in S \mid v \in [\![\mathsf{Init}(\ell)]\!]\}$ is the *initial space*, $S_f = \{(\ell, v) \in S \mid v \in [\![\mathsf{Final}(\ell)]\!]\}$ is the *final space*, the actions are $\Sigma = \mathsf{Lab} \cup \{\mathsf{time}\}$ (we assume that $\mathsf{time} \notin \mathsf{Lab}$) and the transition relation $\rightarrow$ contains all the tuples $((\ell, v), \sigma, (k, w))$ such that:

- (discrete transition) either there exists $e = (\ell, \sigma, k) \in \mathsf{Edg}$ such that $(v, w) \in [\![\mathsf{Jump}(e)]\!]$, or
- (continuous transition) $\ell = k$, $\sigma = \mathsf{time}$ and there exists an $r \in \mathbb{R}^{\geq 0}$ and a continuously differentiable function $\xi : [0, r] \rightarrow \mathbb{R}^X$ such that $\xi(0) = v$, $\xi(r) = w$ and $(\xi(t), \dot{\xi}(t)) \in [\![\mathsf{Flow}(\ell)]\!]$ for all $t \in [0, r]$ and $\xi(t) \in [\![\mathsf{Inv}(\ell)]\!]$ for all $t \in [0, r]$. We call ξ a *trajectory* from v to w. We also write $(\ell, v) \xrightarrow{r} (k, w)$ to emphasize that the continuous transition is of duration r. Usually $\mathsf{Flow}(\ell)$ is a differential equation, in which case ξ is a solution of that differential equation.

We write $(\ell, v) \xrightarrow{\sigma} (k, w)$ if $\rightarrow$ contains the tuple $((\ell, v), \sigma, (k, w))$.

A state $q = (\ell, v) \in S$ is *reachable* if there exists a finite path $q_0\sigma_0q_1\sigma_1 \ldots \sigma_{n-1}q_n$ where $q_0 \in S_0$, $q = q_n$, and $(q_i, \sigma_i, q_{i+1}) \in \rightarrow$ for all $0 \leq i < n$. This path *generates* the word $\bar{\sigma} = \sigma_0\sigma_1 \ldots \sigma_{n-1} \in \Sigma^*$. If $q \in S_f$ is final, we say that the word $\bar{\sigma}$ is *accepted* by H. The set of words that are accepted by H is the *language* of H, denoted $\mathcal{L}(H)$. The set of reachable states of $[\![H]\!]$ is denoted by $\mathsf{Reach}([\![H]\!])$. The transition system $[\![H]\!]$ is *safe* if $\mathsf{Reach}([\![H]\!]) \cap S_f = \emptyset$.

Safety Verification Problem. Many verification problems for hybrid systems reduce to the *safety problem* for hybrid automata.

Definition 8 (Safety verification problem for hybrid automata) Given a hybrid automaton H, the *safety verification problem for hybrid automata* asks whether $[\![H]\!]$ is safe.

A *parameter* in a hybrid automaton is a variable which has first derivative 0 in every location and is never modified by discrete transitions. The *parametric safety*

verification problem for hybrid automata asks, given a hybrid automaton H and a parameter p in H, whether there exists a value $v_p \in \mathbb{R}$ such that $[\![H_{p=v_p}]\!]$ is safe, where $H_{p=v_p}$ is obtained by replacing every constraint φ in H by $\varphi \wedge (p = v_p)$.

Remark There would be no loss of generality in assuming that there is a location ℓ_{bad} such that $\mathsf{Final}(\ell_{\mathsf{bad}}) = \mathsf{true}$ and $\mathsf{Final}(\ell) = \mathsf{false}$ for all $\ell \neq \ell_{\mathsf{bad}}$. Indeed, to reduce any hybrid automaton to this form, it suffices to add transitions $e_\ell = (\ell, \sigma, \ell_{\mathsf{bad}})$ with $\mathsf{Jump}(e_\ell) = \mathsf{Final}(\ell)$ for each $\ell \in \mathsf{Loc}$, and to substitute $\mathsf{Final}(\ell)$ with false for each $\ell \neq \ell_{\mathsf{bad}}$.

30.3 Decidability and Undecidability Results

We review the most important results about the decidability of the safety verification problem for subclasses of hybrid automata. Details and proofs can be found in the given references.

Safety Verification Problem. The safety verification problem is decidable only for restricted classes of hybrid automata. The main classes for which safety verification is decidable are timed automata (see Chap. 29), initialized rectangular automata, and o-minimal hybrid automata [108]. The safety verification problem is undecidable already for the class of rectangular hybrid automata (and therefore also for linear, and affine hybrid automata).

A rectangular predicate over X is an expression of the form $a \prec x \prec b$ where $x \in X$, $\prec \in \{\leq, <\}$, and $a \leq b$ define a nonempty (possibly unbounded) interval with endpoints $a, b \in \mathbb{Q} \cup \{-\infty, \infty\}$. *Rectangular hybrid automata* are hybrid automata where (i) the flow constraint in each location ℓ is a conjunction of rectangular predicates over $\dot{X}$, (ii) the initial, final, and evolution domain conditions are conjunctions of rectangular predicates over X, and (iii) the jump condition of every edge is a conjunction of rectangular predicates over X^+ and expressions of the form $x^+ = x$ for $x \in X$. A hybrid automaton is *initialized* if for every edge $e = (\ell, \sigma, k)$ and for every variable x such that $\{v(\dot{x}) \mid v \in [\![\mathsf{Flow}(\ell)]\!]\} \neq \{v(\dot{x}) \mid v \in [\![\mathsf{Flow}(k)]\!]\}$, it holds that the set $\mathsf{update}_e^x(v) = \{w(x^+) \mid (v, w) \in [\![\mathsf{Jump}(e)]\!]\}$ does not depend on the valuation v (i.e., $\mathsf{update}_e^x(v) = \mathsf{update}_e^x(v')$ for all valuations v, v'). In words, whenever the flow condition is changed for a variable x by a discrete transition e, then this variable is (nondeterministically) reinitialized to a new value in update_e^x that is independent of the previous value.

The following decidability result is obtained by a translation of initialized rectangular hybrid automata to timed automata, preserving safety (see also Sect. 30.5.1).

Theorem 1 ([96]) *The safety verification problem is decidable for initialized rectangular hybrid automata* (*and therefore also for timed automata*).

The safety verification problem remains decidable for various extensions of timed automata. For instance, if *diagonal constraints* of the form $x - y \bowtie c$ for

$x, y \in X, \bowtie \in \{<, \leq, =, >, \geq\}$, and $c \in \mathbb{Q}$ are allowed in guards, or if *assignments* of the form $x^+ = y$ are allowed in updates, then the safety verification problem is still decidable [11, 33]. The decidability result for safety verification, useful for model checking, can be extended to the controller synthesis problem, solved as a game. We refer to Chap. 29 for games on timed automata, and mention the decidability of discrete-time control for rectangular hybrid automata [95].

The safety verification problem becomes undecidable for automata with rectangular flow constraints.

Theorem 2 ([96]) *The safety verification problem is undecidable for rectangular hybrid automata* (*and therefore also for linear, and affine hybrid automata*).

Note that the class of initialized rectangular hybrid automata (for which safety verification is decidable) have a finite language-equivalence quotient [94, 156]. The special case of initialized rectangular hybrid automata with only two variables even has a finite similarity quotient [94], and the class of timed automata has finite bisimilarity quotient (see also Chap. 29).

The result of Theorem 2 has been refined in several directions [96]. The problem is undecidable even if there is a single variable x with two different slopes, i.e., there exist $k_1, k_2 \in \mathbb{Q}$ with $k_1 \neq k_2$ such that in every location ℓ, either $\mathsf{Flow}(\ell)$ implies $\dot{x} = k_1$, or $\mathsf{Flow}(\ell)$ implies $\dot{x} = k_2$. The undecidability result holds for all fixed rational constants $k_1 \neq k_2$. The problem is also undecidable if diagonal constraints or assignments of the form $x^+ = y$ are allowed, and one variable has slope $k \neq 1$. There are extremely simple classes of hybrid systems, stopwatch automata, i.e., timed automata with only differential equations of the form $\dot{x} = 1$ and $\dot{x} = 0$, that are already undecidable [40] (see also Chap. 29). The variant of time-bounded safety verification asks whether, given a time bound T, there exists a final state reachable within a total duration of T time units. This problem is also undecidable for general rectangular hybrid automata, but it is decidable for a larger class than plain safety verification, namely for rectangular hybrid automata with monotone dynamics (the rate of every variable is either always non-negative, or always non-positive [36]).

Note that while differential equations define single continuous executions, the safety verification problem has been considered under various perturbed semantics with finite precision where drifting executions or tubes of executions are considered. It turns out that the undecidability result of Theorem 2 is mostly robust [97], but some decidability results can be obtained [23, 57, 64, 155].

Systems between timed and hybrid automata may remain decidable, e.g., weighted timed automata [15, 25]. Even systems with piecewise constant derivatives quickly become undecidable for dimension three [21]. On the other hand, if the discrete and the continuous parts of a hybrid system are completely independent of each other, the system falls apart into separate continuous systems, so that reachability becomes decidable for certain classes of linear differential equations [108].

Parametric Safety Verification Problem. If parameters are allowed only in the jump conditions of the edges, then it can be shown that the parametric safety veri-

fication problem is decidable for timed automata with one clock [14, 124], and undecidable for timed automata with one parameter and (i) three clocks (all of which being possibly constrained by the parameter) [124], or (ii) four clocks, but only one is compared with the parameter [124]. These undecidability results require the use of equalities in jump conditions. An undecidability result is known for *open* timed automata (in which all guards are open sets, thus forbidding equality constraints) with two parameters and five clocks (among which two are compared with the parameters) [59]. If parameters are allowed in the flow constraints, then it can be shown that the parametric safety verification problem is undecidable for rectangular automata with three variables and one parameter [181].

Computability and Polynomial Constraints. A frequent misconception about the definition of hybrid automata is that they should allow an arbitrary subset $\mathsf{Init}(\ell) \subseteq \mathbb{R}^n$ of the real numbers as initial region for each location ℓ, an arbitrary subset $\mathsf{Inv}(\ell) \subseteq \mathbb{R}^n$ as evolution domain restriction, arbitrary relation $\mathsf{Flow}(\ell) \subseteq \mathbb{R}^n \times \mathbb{R}^n$ as flow constraints, arbitrary relation $\mathsf{Jump}(e) \subseteq \mathbb{R}^n \times \mathbb{R}^n$ jump conditions, and an arbitrary subset $\mathsf{Final}(\ell) \subseteq \mathbb{R}^n$ as final conditions. Generalizations like these have been suggested in the literature numerous times. They are useful as mathematical models, but not for any computational or verification purpose. It is important to understand why.

We can only obtain meaningful model-checking results for a hybrid automaton if we can describe the hybrid automaton (e.g., as an input file in a computer for the model checker). There is no way to describe arbitrary sets $\mathsf{Init}(\ell)$, $\mathsf{Inv}(\ell) \subseteq \mathbb{R}^n$, $\mathsf{Jump}(e) \subseteq \mathbb{R}^n \times \mathbb{R}^n$, etc. as inputs, because there are uncountably many such sets, but model checkers accept only finite input files from a countable set of inputs.

Moreover, even for cases where there is some description of those sets, we still need to equip the model checker with a way to decide membership in those sets. Suppose some model-checking algorithm worked hard to find out that the hybrid automaton will be unsafe when started in a particular state $\nu \in \mathbb{R}^n$. Then, the model checker still needs to find out whether the hybrid automaton allows ν as an initial state or not. That is, we need to give the model checker a way of deciding whether $\nu \in \mathsf{Init}(\ell)$ for any location $\ell \in \mathsf{Loc}$. Mathematically, this is a simple set inclusion and looks trivial. But that does not mean there is a computer program that can decide whether $\nu \in \mathsf{Init}(\ell)$ or $\nu \notin \mathsf{Init}(\ell)$. For arbitrary sets $\mathsf{Init}(\ell) \subseteq \mathbb{R}^n$, this is impossible by classical results on the limits of computation due to Turing, Church, Gödel, and others. The Mandelbrot set is an example of such a set $\mathsf{Init}(\ell)$ for which it is impossible to decide membership even in a very strong model of real computation [32].

Similar observations hold for all the other parts of hybrid automata. Consequently, we have to assume more structure on $\mathsf{Init}(\ell)$ and all the parts of the definitions of hybrid automata. This is the reason why it is crucial that Definition 6 requires hybrid automata to be described in a definable way. Definition 6 requires hybrid automata to be described by polynomial constraints with rational coefficients, which are representable on a computer, unlike constraints with arbitrary real coefficients. This also explains why it is critical to restrict polyhedra to rational coefficients in Definition 5.

It should be noted that these observations about the requirements on hybrid automata are crucial for all model checkers, whether they try to decide fragments or semidecide fragments or whether they just strive to approximately answer the reachability problem. Fundamental limits of computation that represent the *numerical analogue of the halting problem* otherwise cause strong undecidabilities even for approximate answers [147], unless additional assumptions are imposed on the hybrid automata [51, 147].

30.4 Set-Based Reachability Analysis

There are two kinds of events that can take place in a hybrid automaton: time can pass with the state evolving according to the flow constraints, or a jump can take the system instantaneously to a new state. Starting from the initial states, *set-based reachability analysis* exhaustively computes the successor states for both time elapse and jumps in alternation until this no longer produces any new states. Since this process might not terminate (see decidability results in Sect. 30.3), an a priori limit on the search depth is sometimes imposed. The search depth is usually counted in the number of jumps and, in analogy to discrete automata, this is referred to as *bounded model checking*.

Reachability computation can be seen as a generalization of *numerical simulation*. In numerical simulation, one picks an initial state and tries to compute a successor state that lies on one of the solutions of the corresponding flow constraint and also satisfies one of the jump conditions (some intermediate points along the trajectory are usually kept as well). Then one picks one of the successor states of the jump and repeats the process. Like numerical simulation, reachability analysis directly follows the transition semantics of hybrid automata (Definition 7), but considers sets of states instead of single states.

Just like numerical simulation, reachability computation has to use approximations if the dynamics of the system are complex. Working with sets instead of points, approximate reachability can be conservative in the sense that the computed sets are sure to cover all solutions. Computation costs generally increase sharply in terms of the number of continuous variables. Scalable approximations are available for certain types of dynamics, as discussed later in this section, but this performance comes at a price in accuracy. The trade-off between runtime and accuracy remains a central problem in reachability analysis. Surveys of reachability techniques for hybrid automata can be found, e.g., in [7, 117, 119, 165].

30.4.1 Reachability Algorithm

The standard method to compute the reachable states is to iterate the following *one-step successor* operators for discrete and continuous transitions. Given a set

of states S, let $\mathsf{Post}_C(S)$ be the set of states reachable by letting time elapse from any of the states in S,

$$\mathsf{Post}_C(S) = \big\{(\ell, w) \;\big|\; \exists (\ell, v) \in S : (\ell, v) \xrightarrow{\mathsf{time}} (\ell, w)\big\}.$$

Let $\mathsf{Post}_D(S)$ be the set of states resulting from taking a discrete transition from any of the states in S,

$$\mathsf{Post}_D(S) = \big\{(k, w) \;\big|\; \exists (\ell, v) \in S, \exists \sigma \in \mathsf{Lab} : (\ell, v) \xrightarrow{\sigma} (k, w)\big\}.$$

The reachable states are obtained by applying $\mathsf{Post}_C(S)$ and $\mathsf{Post}_D(S)$ repeatedly and recording all states that are obtained. The basic algorithm for *forward reachability* computes the following sequence, starting from the initial states:

$$\begin{aligned} R_0 &= \big\{(\ell, v) \;\big|\; v \in [\![\mathsf{Init}(\ell)]\!]\big\}, \\ R_{i+1} &= R_i \cup \mathsf{Post}_C(R_i) \cup \mathsf{Post}_D(R_i) \quad \text{for } i = 0, 1, 2, \ldots. \end{aligned}$$

The algorithm terminates when a fixed point is reached, i.e., when $R_{i+1} = R_i$ for some $i \geq 0$ (note that $R_i \subseteq R_{i+1}$ for all $i \geq 0$). This simple algorithm does not necessarily terminate, even for systems where reachability is decidable. For example, a system with an (unbounded) counter would enter a new state at each iteration such that the fixed point is never reached. Abstraction techniques such as *widening* [22, 86] are used in program analysis to ensure termination, and while they have been applied to hybrid systems with simple dynamics [92] it is difficult to obtain finite-state abtractions for more general cases.

Reachability with Symbolic States. A semi-algorithm used frequently for reachability of hybrid automata is shown as Algorithm 1. The states of the hybrid automaton H are represented by finite sets of *symbolic states* (ℓ, P), where $\ell \in \mathsf{Loc}$ and P is a set of continuous states in a suitable set representation such as polyhedra. The set of states corresponding to such a set $R = \{(\ell_1, P_1), (\ell_2, P_2), \ldots\}$ is

$$[\![R]\!] = \big\{(\ell, v) \;\big|\; \exists (\ell, P) \in R : v \in P\big\}.$$

If H is safe, Algorithm 1 computes the reachable states by iterating one-step successor computations on such a set R, without guarantee of termination. If H is not safe, the procedure will eventually stop when a nonempty intersection of R with the final states is found. A similar semi-algorithm implements the backward approach by iterating a one-step predecessor operator. Other approaches are possible such as mixed forward-backward, where the forward and backward algorithms are executed in an interleaved fashion [92]. All these variations are semi-algorithms since the problem is undecidable.

The one-step successors $\mathsf{Post}_C(S)$ and $\mathsf{Post}_D(S)$ are implemented for symbolic states by enumerating over locations and transitions, respectively, using the following operators. The *continuous successors* of a set of continuous states P in a location

Algorithm 1: A reachability semi-algorithm using symbolic states

Input : A hybrid automaton $H = \langle \mathsf{Loc}, \mathsf{Lab}, \mathsf{Edg}, X, \mathsf{Init}, \mathsf{Inv}, \mathsf{Flow}, \mathsf{Jump}, \mathsf{Final} \rangle$.
Output: If H is safe then SAFE else UNSAFE.
begin
  $Bad \leftarrow \{(\ell, [\![\mathsf{Final}(\ell)]\!]) \mid \ell \in \mathsf{Loc}\}$;
  $R \leftarrow \{(\ell, \mathsf{post}_\ell([\![\mathsf{Init}(\ell)]\!])) \mid \ell \in \mathsf{Loc}\}$;
  $R_{old} \leftarrow \emptyset$;
  while $[\![R]\!] \not\subseteq [\![R_{old}]\!]$ **do**
    $R_{old} \leftarrow R$;
    $R \leftarrow \{(\ell, \mathsf{post}_\ell(\mathsf{post}_\varepsilon(P))) \mid (\ell, P) \in R \wedge \varepsilon = (\ell, \sigma, k) \in \mathsf{Edg}\}$;
    if $[\![R]\!] \cap [\![Bad]\!] \neq \emptyset$ **then return** UNSAFE;
  return SAFE ;
end

ℓ is the set of continuous states

$$\mathsf{post}_\ell(P) = \left\{ x' \;\middle|\; \exists x \in P : (\ell, x) \xrightarrow{\mathsf{time}} (\ell, x') \right\}.$$

Similarly, the *discrete successors* of a set of continuous states P for an edge $\varepsilon = (\ell, \sigma, k)$ is the set of continuous states

$$\mathsf{post}_\varepsilon(P) = \left\{ x' \;\middle|\; \exists x \in P : (\ell, x) \xrightarrow{\sigma} (k, x') \right\}.$$

Formally, the one-step successors of a set of symbolic states R are expressed using the above operators as

$$\mathsf{Post}_C([\![R]\!]) = \left[\!\!\left[\left\{ (\ell, \mathsf{post}_\ell(P)) \;\middle|\; \exists (\ell, P) \in R \right\} \right]\!\!\right],$$
$$\mathsf{Post}_D([\![R]\!]) = \left[\!\!\left[\left\{ (k, \mathsf{post}_\varepsilon(P)) \;\middle|\; \exists (\ell, P) \in R, \varepsilon = (\ell, \sigma, k) \in \mathsf{Edg} \right\} \right]\!\!\right].$$

In the following, we discuss the above successor operators $post_\ell(P)$, $\mathsf{post}_\varepsilon(P)$ for different classes of hybrid automata with increasingly complex continuous dynamics. We will focus mainly on computing time elapse successors, since this operation usually dominates costs. Other operations of the reachability algorithm may also become bottlenecks, e.g., computing the discrete successors, containment checking, and clustering.

30.4.2 Piecewise Constant Dynamics

Hybrid automata with piecewise constant dynamics (PCDA) are a special case of hybrid automata with polynomial dynamics (Definition 6), where all constraints are conjunctivelinear and the flow constraints are linear predicates over dotted variables

only. That is, the derivatives of the variables are independent of the current continuous state. They are also called *linear hybrid automata* (LHA), where the term linear refers to trajectories instead of dynamics (they do not allow the linear dynamics discussed in the next section). In order to avoid possible confusion resulting from this terminology, we prefer the name PCDA.

Definition 9 (Hybrid automaton with piecewise constant dynamics) A hybrid automaton $H = \langle \mathsf{Loc}, \mathsf{Lab}, \mathsf{Edg}, X, \mathsf{Init}, \mathsf{Inv}, \mathsf{Flow}, \mathsf{Jump}, \mathsf{Final} \rangle$ with polynomial dynamics is called a *hybrid automaton with piecewise constant dynamics* iff:

- Init, Inv, Final are conjunctivelinear constraints over X,
- Flow are conjunctivelinear constraints over $\dot{X}$, and
- Jump are conjunctivelinear constraints over $X \cup X^+$.

PCDA are of particular interest in formal verification because the one-step successors can be computed exactly, which is not the case for the more complex dynamics discussed in later sections.

Examples of flow constraints of a PCDA include differential inclusions such as $\dot{x} \in [1, 2]$, and conservation laws such as $\dot{x} + \dot{y} = 0$. The jump constraints of a PCDA admit arbitrary linear updates of the variables, which can generate complex behavior. For example, PCDA can model discrete-time affine systems, a widely used class of control systems, by using jump constraints of the form $x^+ = Ax + b$. Chaotic behavior can arise in PCDA due to switching flows [42] or guarded jumps, with which one can model piecewise affine maps such as the tent map [48].

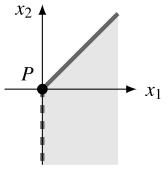
Continuous Successors. In the following, we discuss computing the states reachable by time elapse in a given location ℓ of a PCDA and write x as shorthand for the state (ℓ, x). By definition, a trajectory can be an arbitrarily curved function as long as it is differentiable and satisfies both flow constraints and evolution domain restrictions. For the purposes of reachability, it suffices to consider only straight-line trajectories of PCDA, as formalized in the following lemma.

Lemma 1 ([13]) *In any given location of a PCDA, there is a trajectory $\xi(t)$ from $x = \xi(0)$ to $x' = \xi(r)$ for some $r > 0$ iff $\eta(t) = x + qt$ with $q = \frac{x'-x}{r}$ is a trajectory from x to x'.*

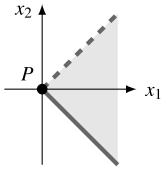
Using this lemma, we now show that the states reachable by time elapse from a polyhedral set of states P are given by the union of P with a polyhedron that is readily computable [8, 28]. Consider polyhedra P and Q. The states on straight-line trajectories starting in P with constant derivative $\dot{x} = q$ for any $q \in Q$ are the *time successors*

$$P \nearrow Q = \left\{ x' \;\middle|\; x \in P, q \in Q, t \in \mathbb{R}^{\geq 0}, x' = x + qt \right\}. \tag{1}$$

We now transform the right-hand term of (1) into a linear constraint. Let P and Q be polyhedra given in vector-matrix form as $P = \{x \mid Ax \bowtie b\}$, $Q = \{q \mid \bar{A}q \bar{\bowtie} \bar{b}\}$.



(a) $P \nearrow Q$ with closed and unbounded $Q = \{q_1 = 1 \wedge q_2 \leq 1\}$



(b) $P \nearrow Q'$ with bounded and non-closed $Q' = \{q_1 = 1 \wedge -1 \leq q_2 < 1\}$

Fig. 3 Given a polyhedron P and a polyhedral set of derivatives Q, the time successors $P \nearrow Q$ can be a convex set that is not a single polyhedron but the union of P with another polyhedron

By separating the case $t = 0$ from $t > 0$ in (1) we have $q = \frac{x'-x}{t}$. Eliminating q and multiplying with t yields

$$P \nearrow Q = P \cup \{x' \mid Ax \bowtie b \ \wedge \ \bar{A}(x' - x) \bar{\bowtie} \bar{b} \cdot t \ \wedge \ t > 0\}. \tag{2}$$

The right-hand term of the union in (2) is a polyhedron that can be computed by quantifier elimination over $X \cup \{t\}$ using, e.g., Fourier–Motzkin elimination. If Q is closed and bounded, the constraint $t > 0$ in (2) can be replaced by $t \geq 0$, so the right-hand term contains P and $P \nearrow Q$ becomes a single polyhedron. The following example illustrates that $P \nearrow Q$ can be the union of two polyhedra.

Example 1 For $P = \{x_1 = 0 \wedge x_2 = 0\}$, and Q, Q' given in Fig. 3, (2) yields

$$\begin{aligned} P \nearrow Q &= P \cup \{(x_1', x_2') \mid x_1 = 0 \wedge x_2 = 0 \wedge x_1' - x_1 = t \wedge x_2' - x_2 \leq t \wedge t > 0\} \\ &= P \cup \{(x_1', x_2') \mid x_1' = t \wedge x_2' \leq t \wedge t > 0\} \\ &= P \cup \{(x_1', x_2') \mid x_1' > 0 \wedge x_2' \leq x_1'\}. \end{aligned}$$

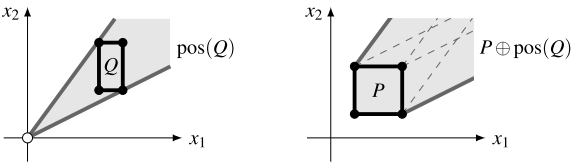
Here, the closed but unbounded set Q results in a convex set $P \nearrow Q$ that is not a polyhedron but the union of two polyhedra. Similarly, the bounded but non-closed set Q' results in $P \nearrow Q' = P \cup \{x_1' > 0 \wedge -x_1' \leq x_2' < x_1'\}$, which is also convex and not a polyhedron.

The time successor operation can also be carried out using geometrical operations on the polyhedra P and Q as shown in Fig. 4 [86]. The *positive cone* of Q is the polyhedral set $\text{pos}(Q) = \{q \cdot t \mid q \in Q, t > 0\}$. The time successors are given by the Minkowski sum[3] of P and the positive cone of Q,

$$P \nearrow Q = P \cup (P \oplus \text{pos}(Q)). \tag{3}$$

If P and Q are closed with generator representation (V, R) and (V', R'), respectively, then a generator representation of $P \nearrow Q$ is $(V, R \cup V' \cup R')$.

[3]The *Minkowski sum* is defined as $P \oplus Q = \{p + q \mid p \in P, q \in Q\}$.



(a) Q and its cone $\mathrm{pos}(Q)$ (b) P and $P \nearrow Q = P \cup P \oplus \mathrm{pos}(Q)$

Fig. 4 The time successors $P \nearrow Q$ using geometric operations on polyhedra P and Q

It remains to ensure that the time successors are reachable by trajectories that satisfy $\mathsf{Inv}(\ell)$. Assuming that $P \subseteq [\![\mathsf{Inv}(\ell)]\!]$, this restriction reduces to $x' \in [\![\mathsf{Inv}(\ell)]\!]$ since $[\![\mathsf{Inv}(\ell)]\!]$ is convex and only straight line trajectories need to be considered. This leads us to the following discrete successor operator for PCDA.

Lemma 2 ([8]) *The* continuous successors *of a polyhedron P in a location ℓ of a PCDA H is the set*:

$$\mathsf{post}_\ell(P) = \left(P \nearrow [\![\mathsf{Flow}(\ell)]\!]\right) \cap [\![\mathsf{Inv}(\ell)]\!].$$

Discrete Successors. The *discrete successors* of a polyhedron P for an edge $\varepsilon = (\ell, \sigma, k)$ of a PCDA H is the set:

$$\mathsf{post}_\varepsilon(P) = \left\{x^+ \mid \exists x \in P : \left(x, x^+\right) \in [\![\mathsf{Jump}(\varepsilon)]\!] \wedge x^+ \in [\![\mathsf{Inv}(k)]\!]\right\}.$$

This set is defined using existential quantification, and computing it may require costly quantifier elimination. Frequently occurring special cases can be computed more efficiently. As an example, consider $\mathsf{Jump}(e)$ given by a guard $x \in G$ and a reset $x^+ = Cx + d$, with a constant matrix C and a vector d of appropriate dimensions. The discrete successors are

$$\mathsf{post}_\varepsilon(P) = \left(C(P \cap G) \oplus \{d\}\right) \cap [\![\mathsf{Inv}(k)]\!]. \tag{4}$$

If C is invertible and all sets are polyhedra in constraint representation, the computation is straightforward since intersection corresponds to concatenation of constraints, and for any polyhedron $Q = \{x \mid Ax \bowtie b\}$,

$$CQ \oplus \{d\} = \left\{x \mid AC^{-1}x \bowtie b + C^{-1}d\right\}.$$

Computational Cost. Computing the continuous successors using (3) involves the cone, Minkowksi sum, and intersection operations, for details see [22, 86]. The cone and Minkowski sum are efficient only in the generator representation of a polyhedron (see Definition 5). The intersection operation is efficient only in constraint representation. Translating the polyhedron from constraints to generators and vice versa can produce a number of generators that is exponential in the number of variables. For example, consider that an n-dimensional cube has $2n$ constraints and 2^n vertices. Dually, an n-dimensional *cross-polytope* (hyperoctahedron) has $2n$ vertices and 2^n constraints. In total, the cost of computing the continuous successors is exponential in the number of variables. Tools such as HyTech and PHAVer use the

geometric version (3) of the time successor operator since in practice it is often more efficient than quantifier elimination [87]. The operator is available in computational geometry libraries such as the Parma Polyhedra Library (PPL) [22].

The cost of computing the discrete successors is exponential for polyhedra in constraint representation since it involves quantifier elimination. For some frequently occurring special cases the cost is polynomial, e.g., in the case of (4) with invertible map.

The containment and emptiness tests in Algorithm 1 are carried out pairwise over the elements of sets of symbolic states. The *containment test* $P \subseteq Q$ is solvable with linear programming (and thus in polynomial time) if P, Q are in constraint representation.[4] The *emptiness test* $P = \emptyset$ is solvable as a linear program if P is in constraint representation and trivial if P is in generator representation.

Path Constraints. A *path* of a hybrid automaton is a sequence of adjacent edges (usually from an initial to a final location). An interesting property of PCDA is that the reachable states along a given path can be encoded by a conjunction of linear constraints, the so-called *path constraints*. The reachability problem for a given path can therefore be solved very efficiently using linear programming. This approach has been implemented in the tool BACH [37]. The number of paths in a PCDA can be infinite if there are cycles, so techniques such as CEGAR have been used to reduce the number of paths to be checked and accelerate termination [162].

30.4.3 Piecewise Affine Dynamics

Hybrid automata with piecewise affine dynamics (PWA) are a special case of hybrid automata with polynomial dynamics (Definition 6), where all constraints are linear and the flow constraints are linear ordinary differential equations (ODEs). We divide the continuous variables into *state variables* $X = \{x_1, \ldots, x_n\}$, whose derivative is explicitly defined, and *input variables* $U = \{u_1, \ldots, u_m\}$, whose derivative is unconstrained. The input variables can be used to model nondeterminism such as open inputs to the system, approximation errors, disturbances, etc.

In each location of a PWA, the continuous dynamics are *affine*, i.e., given by differential equations of the form

$$\dot{x} = Ax + Bu, \quad u \in \mathcal{U}, \tag{5}$$

where A and B are matrices of appropriate dimension and the *input set* $\mathcal{U}$ is compact and convex. Note that differential inclusions like $\dot{x} \in \mathcal{U}$ and $Ax - b \leq \dot{x} \leq Ax + b$ can be brought into this form by introducing auxiliary variables. Similarly, jump constraints of an edge e define resets of the form

$$x^+ = Cx + Du, \tag{6}$$

[4] Checking $P \subseteq Q$ is polynomial unless P is in constraint representation and Q is in generator representation, in which case it is known to be NP-complete [71].

where x^+ denotes the value of x after the jump, u is defined as above and C and D are matrices of appropriate dimension. The jump constraints also define a set $\mathcal{G}$ called the *guard* of the edge, and a jump can only take place if $x \in \mathcal{G}$. The formal definition of PWA is as follows.

Definition 10 (Hybrid automaton with piecewise affine dynamics) A *hybrid automaton with piecewise affine dynamics* is a hybrid automaton $H = \langle \mathsf{Loc}, \mathsf{Lab}, \mathsf{Edg}, X \cup U, \mathsf{Init}, \mathsf{Inv}, \mathsf{Flow}, \mathsf{Jump}, \mathsf{Final} \rangle$ where

- Init and Inv are conjunctivelinear constraints over X.
- Inv are conjunctivelinear constraints over $X \cup U$, such that each linear term ranges over variables exclusively from either X or U (no correlation between state and input variables). The input set $\mathcal{U}$ of a location ℓ is given by the terms of $\mathsf{Inv}(\ell)$ that range over U and must be closed and bounded.
- Flow are constraints over $\dot{X} \cup X \cup U$ of the form $\dot{x} = Ax + Bu$.
- Jump are conjunctivelinear constraints over $X^+ \cup X \cup U$ whose terms either range over X or are of the form $x^+ = Cx + Du$. The *guard* set $\mathcal{G}$ of an edge e is given by the terms of $\mathsf{Jump}(e)$ that range over X.

The reachable states of a PWA can be computed using Algorithm 1 from Sect. 30.4.1, with suitable operators post_ℓ for continuous and post_e for discrete successors that will be presented in the following section.

30.4.3.1 Successor Computations

The successor computations for affine dynamics can be approximated by sequences of geometric set operations. We first present such a sequence for the continuous successors, then give the equation for the discrete successors. Different set representations can be used to implement these operations, and a selection are discussed in the subsequent Sect. 30.4.3.2.

Continuous Successors. In the following, we discuss how to compute the states reachable by time elapse in a given location ℓ. Since ℓ is clear from the context we call x a (continuous) state. We will initially ignore any evolution domain restriction on x and discuss it after the basic construction has been presented. The evolution of the input variables is described by an *input signal* $\zeta : \mathbb{R}^{\geq 0} \to \mathcal{U}$ that attributes to each point in time a value of the input u. The input signal does not need to be continuous. A trajectory $\xi(t)$ from a state x_0 is the solution of the differential equation (5) for initial condition $\xi(0) = x_0$ and a given input signal ζ. It has the form

$$\xi_{x_0,\zeta}(t) = e^{At}x_0 + \int_0^t e^{A(t-s)}B\zeta(s)ds. \tag{7}$$

It consists of the superposition of the solution of the *autonomous* system, obtained for $\zeta(t) = 0$, and the input integral obtained for $x_0 = 0$. In the following, this decomposition of (7) will be exploited to obtain efficient and accurate approximations.

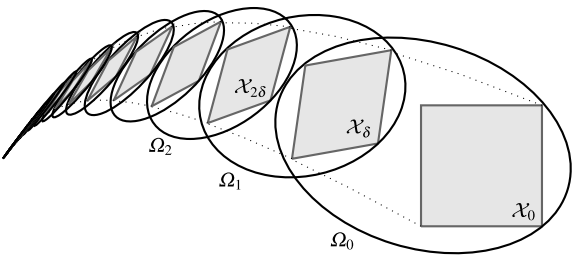


Fig. 5 A sequence of sets $\Omega_0, \Omega_1, \ldots$ that covers $\mathcal{X}_t$ over a finite time horizon T. The choice of set representation for Ω_k (illustrated here by *ellipsoids*) has a substantial impact on accuracy and computational complexity

A state x' is reachable from some initial set of states $\mathcal{X}_0$ in time t if for some $x_0 \in \mathcal{X}_0$ and some ζ, $x' = \xi_{x_0,\zeta}(t)$. We now describe the reachable states as sets using (7). Let $\mathcal{X}_t$ be the states reachable in time t from any state in $\mathcal{X}_0$ and let $\mathcal{Y}_t$ be the states reachable from $\mathcal{X}_0 = \{0\}$, then (7) can be written as

$$\mathcal{X}_t = e^{At}\mathcal{X}_0 \oplus \mathcal{Y}_t. \tag{8}$$

The goal is to conservatively approximate the reachable states over some finite time horizon T, i.e., to compute a finite sequence of sets $\Omega_0, \Omega_1, \ldots$ such that

$$\bigcup_{0 \le t \le T} \mathcal{X}_t \subseteq \Omega_0 \cup \Omega_1 \cup \ldots. \tag{9}$$

We present the construction of a sequence of Ω_k for a fixed *sampling time* $\delta > 0$ such that Ω_k covers $\mathcal{X}_t$ for $t \in [k\delta, (k+1)\delta]$, as illustrated in Fig. 5. The so-called *semi-group* property of reachability says that, starting from $\mathcal{X}_s$, for any $s \ge 0$, and then waiting r time units leads to the same states as starting from $\mathcal{X}_0$ and waiting $r + s$ time units. Applying this to (8), we obtain that for any $r, s \ge 0$,

$$\mathcal{X}_{r+s} = e^{Ar}\mathcal{X}_s \oplus \mathcal{Y}_r. \tag{10}$$

Substituting $r \leftarrow \delta$, $s \leftarrow k\delta$, we obtain a recursive time discretization in the form of

$$\mathcal{X}_{(k+1)\delta} = e^{A\delta}\mathcal{X}_{k\delta} \oplus \mathcal{Y}_\delta.$$

It follows that if we have initial approximations Ω_0 and Ψ_δ such that

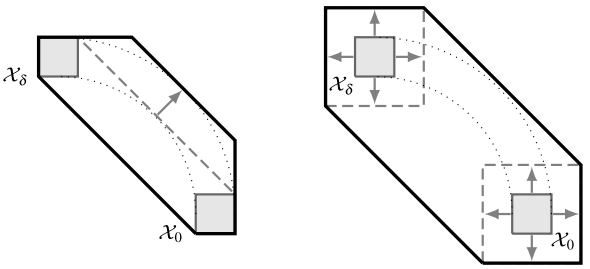
$$\bigcup_{0 \le t \le \delta} \mathcal{X}_t \subseteq \Omega_0, \qquad \mathcal{Y}_\delta \subseteq \Psi_\delta, \tag{11}$$

then the sequence

$$\Omega_{k+1} = e^{A\delta}\Omega_k \oplus \Psi_\delta \tag{12}$$

satisfies (9). Note that Ω_0 covers the reachable set over an interval of time $[0, \delta]$, while Ψ_δ covers the values of the input integral at a single time instant δ.

Computing Initial Approximations Ω_0 and Ψ_δ. The set Ω_0 needs to cover $\mathcal{X}_t$ from $t = 0$ to $t = \delta$. A good starting point for such a cover is the convex hull of



(a) pushing facets (b) bloating

Fig. 6 An approximation Ω_0 that covers $\mathcal{X}_t$ for $t \in [0, \delta]$ can be obtained from the convex hull of $\mathcal{X}_0$ and $\mathcal{X}_\delta$ by enlarging it enough to compensate for the curvature of trajectories

$\mathcal{X}_0$ and $\mathcal{X}_\delta$. One approach, shown in Fig. 6(a), is to compute the convex hull in constraint representation, and push the facets out far enough to be conservative [81]. The required values can be computed from a Taylor approximation of (7) [19], or by solving an optimization problem [45]. Note that the cost of computing the exact constraints of the convex hull can be exponential in the number of variables, which limits the scalability of this approach.

A scalable way to obtain Ω_0 is to bloat $\mathcal{X}_0$ and $\mathcal{X}_\delta$ enough to compensate for the curvature of trajectories [75], as illustrated in Fig. 6(b). We present the approach from [75], which uses uniform bloating and whose approximation error is asymptotically linear in the time step δ as $\delta \to 0$. This is asymptotically optimal for any approximation containing the convex hull of $\mathcal{X}_0$ and $\mathcal{X}_\delta$ [110]. The bloating can be made non-uniform in space and time to obtain a more precise approximation [68, 110]. The bloating factor is derived from a Taylor approximation of (7), whose remainder is bounded using norms. To formalize the above statements, we use the following notation. Let $\|\cdot\|$ be a vector norm and let $\|A\|$ be its induced matrix norm.[5] Let $\mu(\mathcal{X}) = \max_{x \in \mathcal{X}} \|x\|$ and let $\mathcal{B}$ be the unit *ball* of the norm, i.e., the largest set $\mathcal{B}$ such that $\mu(\mathcal{B}) = 1$. For a scalar c, let $c\mathcal{X} = \{cx \mid x \in \mathcal{X}\}$. The approximation error is measured using the *Hausdorff distance* between sets $\mathcal{X}, \mathcal{Y}$,

$$d_H(\mathcal{X}, \mathcal{Y}) = \max\left\{ \sup_{x \in \mathcal{X}} \inf_{y \in \mathcal{Y}} \|x - y\|, \sup_{y \in \mathcal{Y}} \inf_{x \in \mathcal{X}} \|x - y\| \right\}.$$

Lemma 3 ([75]) *Given a set of initial states $\mathcal{X}_0$ and affine dynamics (5), let*

$$\alpha_\delta = \mu(\mathcal{X}_0) \cdot \left(e^{\|A\|\delta} - 1 - \|A\|\delta\right),$$
$$\beta_\delta = \frac{1}{\|A\|} \mu(B\mathcal{U}) \cdot \left(e^{\|A\|\delta} - 1\right),$$
$$\Omega_0 = \mathrm{chull}\left(\mathcal{X}_0 \cup e^{A\delta}\mathcal{X}_0\right) \oplus (\alpha_\delta + \beta_\delta)\mathcal{B},$$
$$\Psi_\delta = \beta_\delta \mathcal{B}.$$

[5] For example, the *infinity norm* $\|x\|_\infty = \max\{|x_1|, \ldots, |x_n|\}$ induces the matrix norm $\|A\| = \max_{1 \le i \le n} \sum_{j=1}^{m} |a_{ij}|$, where A is of dimension $n \times m$. Its ball $\mathcal{B}_\infty$ is a cube of side length 2.

Then $\bigcup_{0\le t\le\delta}\mathcal{X}_t \subseteq \Omega_0$ *and* $\mathcal{Y}_\delta \subseteq \Psi_\delta$. *Furthermore, if* $B\mathcal{U}$ *is a ball of the norm, i.e.,* $B\mathcal{U} = \mu(B\mathcal{U})\mathcal{B}$, *the approximation error is bounded by*

$$d_H\Big(\bigcup_{0\le t\le\delta}\mathcal{X}_t, \Omega_0\Big) \le \delta e^{\|A\|\delta}\big(\mu(B\mathcal{U}) + \big(\tfrac{1}{2}+\delta\big)\|A\|\mu(\mathcal{X}_0)\big),$$
$$d_H(\mathcal{Y}_\delta, \Psi_\delta) \le \delta^2\|A\|e^{\|A\|\delta}\mu(B\mathcal{U}).$$

Propagating the initial approximation Ω_0 forward in time using (12) gives an approximation of $\mathcal{X}_t$ over a bounded horizon. The following theorem gives a bound on the total approximation error.

Theorem 3 ([75]) *Given* Ω_0 *and* Ψ_δ *as defined in Lemma 3, let* $\Omega_{k+1} = e^{A\delta}\Omega_k \oplus \Psi_\delta$ *for* $k = 1, \dots, N-1$. *Then* $\bigcup_{0\le t\le N\delta}\mathcal{X}_t \subseteq \bigcup_{0\le k\le N-1}\Omega_k$. *Furthermore, if* $B\mathcal{U}$ *is a ball of the norm, the approximation error is bounded by*

$$d_H\Big(\bigcup_{0\le t\le N\delta}\mathcal{X}_t, \bigcup_{0\le k\le N-1}\Omega_k\Big) \le \delta e^{\|A\|N\delta}\left(2\mu(B\mathcal{U}) + \left(\frac{1}{2}+\delta\right)\|A\|\mu(\mathcal{X}_0)\right).$$

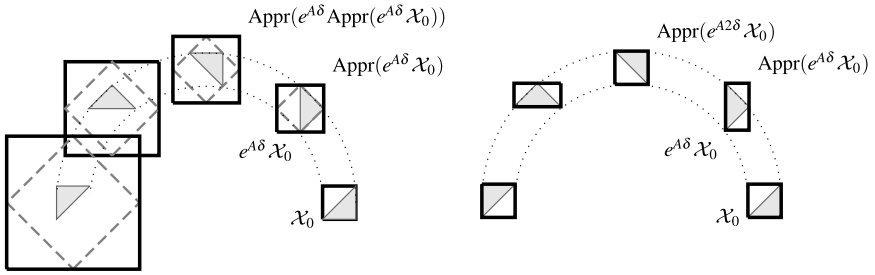
Approximations and the Wrapping Effect. The sequence in (12) can be problematic to compute since the complexity of Ω_k may increase sharply with k. We illustrate this for the case where Ω_k is a polytope in generator representation, and a similar argument can be made for constraint representation. Let N_k be the number of vertices of Ω_k and let Ψ_δ have M vertices. Since Ω_{k+1} is the sum of $e^{A\delta}\Omega_k$ with Ψ_δ it can have $N_{k+1} = N_k \cdot M$ vertices. Resolving the recursion, we get the tight upper bound $N_k \le N_0 \cdot M^k$. To avoid this increase in complexity, we approximate each Ω_k by a simplified set. Let Appr be an *approximation function* such that for any set P, $P \subseteq \mathrm{Appr}(P)$. The sequence (12) then becomes

$$\hat{\Omega}_{k+1} = \mathrm{Appr}\big(e^{A\delta}\hat{\Omega}_k \oplus \Psi_\delta\big). \tag{13}$$

For example, if Appr computes the interval hull (bounding box) and Ω_0 is a polytope, then all $\hat{\Omega}_k$ are polytopes with $2n$ facets. However, the recursive application of the approximation function can lead to an exponential increase in the approximation error. This phenomenon is known in numerical analysis as the *wrapping effect* [105] and is illustrated in Fig. 7.

For affine dynamics, the wrapping effect can be avoided by combining two techniques [77]. First, the approximation operator is chosen such that it distributes over Minkowski sum, i.e., $\mathrm{Appr}(P \oplus Q) = \mathrm{Appr}(P) \oplus \mathrm{Appr}(Q)$. This is the case, e.g., for the interval hull (bounding box). Second, the alternation of the map $e^{Ak\delta}$ with the Minkowski sum in (12) is avoided by splitting it into two sequences

$$\begin{aligned}\hat{\Psi}_{k+1} &= \mathrm{Appr}\big(e^{Ak\delta}\Psi_\delta\big) \oplus \hat{\Psi}_k, \quad \text{with } \hat{\Psi}_0 = \{0\},\\ \hat{\Omega}_k &= \mathrm{Appr}\big(e^{Ak\delta}\Omega_0\big) \oplus \hat{\Psi}_k.\end{aligned} \tag{14}$$



(a) approximation with wrapping effect (b) using a wrapping-free algorithm

Fig. 7 The wrapping effect can lead to an exponential increase in the approximation error that can be avoided for affine dynamics. This example shows the exact solution $e^{Ak\delta}\mathcal{X}_0$ (*shaded*) and an interval hull approximation (*thick*), with $e^{A\delta}$ performing a rotation of 45 degrees around the origin. The wrapping effect occurs if the approximation is applied to the map of the previous approximation (*dashed*). To illustrate the effect more clearly, $\mathcal{X}_0$ is used here instead of Ω_0

For sequence (14) it holds that $\hat{\Omega}_k = \mathrm{Appr}(\Omega_k)$, which means the resulting approximation is free of the wrapping effect. The total approximation error consists of the bounds of Theorem 3 plus the error introduced by the operator Appr (measured in terms of the Hausdorff distance).

The approach is easily extended to variable time steps by adapting Ω_0 and Ψ_δ to the time step while computing the sequence [68].

Evolution Domain Restriction. So far we have neglected the evolution domain restriction (invariant) $\mathsf{Inv}(\ell)$ of the location. Let $\mathcal{S} = [\![\mathsf{Inv}(\ell)]\!]$. A simple but efficient heuristic tries to find, if it exists, the smallest K such that Ω_K lies completely outside $\mathcal{S}$. The search for such a K may be combined with finding a suitable time horizon T and a suitable time step δ (this search obviously might not terminate). Then one computes the sequence $\Omega_0, \ldots, \Omega_K$ and obtains the sequence $\bar{\Omega}_k = \Omega_k \cap \mathcal{S}$ as an approximation of the continuous successors over the time horizon $T = K\delta$.

In cases where the above solution is overly conservative, one can improve the approximation using the following approach from [83]. Let $\mathcal{S}_t$ be the states reachable from $\mathcal{S}$ (neglecting the evolution domain restriction), and let $\xi(\tau)$ be a trajectory inside $\mathcal{S}$ for all $0 \le \tau \le t$. Then the semi-group property implies that $\xi(\tau + s) \in \mathcal{S}_s$ for all $0 \le s \le t - \tau$, so that $\xi(t) \in \bigcap_{0 \le \tau \le t} \mathcal{S}_\tau$. We may therefore improve the approximation by intersecting Ω_k with an approximation of the states reachable from $\mathcal{S}$, which we obtain from the sequence in (14) with $\Omega_0 \leftarrow \mathcal{S}$. This leads to the following sequence $\bar{\Omega}_k$ that approximates the continuous successors, starting with $k = 0$ and $\Psi_0 = \{0\}$:

$$\begin{aligned} \Psi_{k+1} &= \mathrm{Appr}\big(e^{Ak\delta}\Psi_\delta\big) \oplus \Psi_k, \\ \bar{\Omega}_k &= \big(\mathrm{Appr}\big(e^{Ak\delta}\Omega_0\big) \oplus \Psi_k\big) \cap \bigcap_{0 \le i \le k} \big(\mathrm{Appr}\big(e^{Ai\delta}\mathcal{S}\big) \oplus \Psi_i\big). \end{aligned} \tag{15}$$

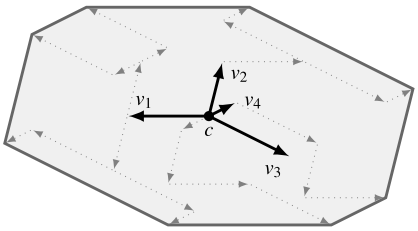


Fig. 8 A zonotope is a special form of centrally symmetric polytope, as illustrated here with generators v_1, v_2, v_3, v_4, and center $c = 0$

Discrete Successors. Consider an edge $\varepsilon = (\ell, \sigma, k)$ of a PWA, whose jump constraints define the reset map

$$x^+ = Cx + Du$$

and the guard set $\mathcal{G}$, which only lets states jump where $x \in \mathcal{G}$. Recall that $u \in \mathcal{U}$, where $\mathcal{U}$ is compact, convex, and given by constraints in $Inv(\ell)$. Let $\mathcal{S}^+ = [\![\mathsf{Inv}(k)]\!]$ be the evolution domain restriction of the target location. The discrete successors of a set P can be written using geometric operators as

$$\mathsf{post}_\varepsilon(P) = \big(C(P \cap \mathcal{G}) \oplus D\mathcal{U}\big) \cap \mathcal{S}^+.$$

We now turn to representing the individual sets in the sequences Ψ_k and Ω_k, and which approximation operator Appr to use.

30.4.3.2 Set Representations

Several set representations have been proposed in the literature for computing the continuous successors under affine dynamics, using variations of the algorithm presented in the previous section. To be efficient, scalable implementations or approximations need to be available for the operators in the algorithm. Using the initial approximation from Lemma 3 and the recurrence equation (14), the operators are linear map, Minkowski sum, convex hull, and intersection. The following paragraphs summarize the results for a selection of prominent representations.

Ellipsoids. The first scalable reachability algorithms for affine dynamics were obtained for ellipsoids, see [107, 133] and references therein. An approximation of the reachable states using ellipsoids is shown in Fig. 5. A nondegenerate *ellipsoid* $\mathcal{E}(c, Q) \subseteq \mathbb{R}^n$ is represented by a center $c \in \mathbb{Q}^n$ and a positive definite[6] matrix $Q \in \mathbb{Q}^{n \times n}$,

$$\mathcal{E}(c, Q) = \big\{x \mid (x - c)^\mathsf{T} Q^{-1}(x - c) \leq 1\big\}$$

(this can be generalized to degenerate ellipsoids). Deterministic affine transforms can be computed efficiently for ellipsoids. For a matrix $A \in \mathbb{Q}^{n \times n}$ and vector

[6] A matrix Q is positive definite iff it is symmetric and $x^\mathsf{T} Q x > 0$ for all $x \neq 0$.

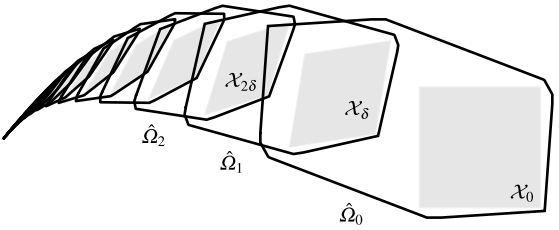


Fig. 9 A reach set cover $\hat{\Omega}_0, \hat{\Omega}_1, \ldots$, computed with zonotopes using the implementation in [3] (*solid*)

$b \in \mathbb{Q}^n$,

$$A\mathcal{E}(c, Q) + b = \mathcal{E}\big(Ac + b, AQA^{\mathsf{T}}\big).$$

Ellipsoids are not closed under Minkowski sum, convex hull, or intersection. Using ellipsoids one therefore generally suffers from the wrapping effect unless $B\mathcal{U}$ is a singleton. Efficient approximations are available for Minkowski sum, convex hull, and special cases of intersection, but the computation of discrete successors can be problematic in terms of accuracy. For an implementation, see [106].

Zonotopes. Zonotopes are a compact representation for a special form of polytopes that have been used successfully for reachability analysis due to their computationally attractive features [3, 75]. A *zonotope* $P \subseteq \mathbb{R}^n$ is defined by a center $c \in \mathbb{Q}^n$ and a finite number of generators $v_1, \ldots, v_k \in \mathbb{Q}^n$ that span the polytope as bounded linear combinations from the center:

$$P = \left\{ c + \sum_{i=1}^{k} \alpha_i v_i \;\middle|\; \alpha_i \in [-1, 1] \right\}.$$

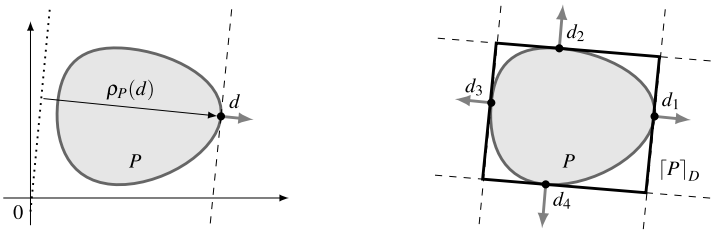
A common denotation for this zonotope is $P = (c, \langle v_1, \ldots, v_k \rangle)$. A zonotope with k generators is an affine transformation of a k-dimensional unit hypercube. Zonotopes are central-symmetric convex polytopes, see Fig. 8 for an illustration. Affine transformations can be computed efficiently for zonotopes. For a matrix $A \in \mathbb{Q}^{m \times n}$, the image of the linear transformation can simply be computed component-wise:

$$AP = \big(Ac, \langle Av_1, \ldots, Av_k \rangle\big)$$

The Minkowski sum can be computed efficiently for zonotopes $P = (c, \langle v_1, \ldots, v_k \rangle)$ and $Q = (d, \langle w_1, \ldots, w_m \rangle)$ by a single vector addition and a single list concatenation:

$$P \oplus Q = \big(c + d, \langle v_1, \ldots, v_k, w_1, \ldots, w_m \rangle\big).$$

Since zonotopes are closed under Minkowski sum, it is straightforward to devise an approximation operator Appr that distributes over Minkowski sum and use the wrapping-free sequence (14). When the list of generators of a zonotope becomes large, one can efficiently compute a smaller list that results in a cover of the original zonotope [75].



(a) support function in direction d (b) outer approximation with $D = \{d_1, d_2, d_3, d_4\}$

Fig. 10 Evaluating the support function in a set of directions gives a polyhedral outer approximation that can be computed very efficiently

Zonotopes are neither closed under convex hull, nor under intersection. Efficient approximations exist, and the accuracy of approximating the convex hull in the above reachability algorithm can be improved by taking smaller time steps. However, the lack of accuracy in intersections can make the computation of discrete successors with zonotopes problematic. In special cases it can be advantageous to use an approach called *continuization* to avoid the intersection operation, see [5]. Instead of intersecting a set of states with the guard set and then applying the dynamics of the successor location to the result, the states suspected to intersect with the guard set (by some approximative measure) are subjected to nondeterministic dynamics that overapproximate the dynamics both before and after the jump. The dynamics of the successor location are used once enough time steps have been carried out to be sure the set no longer intersects with the guard set.

Reachability with zonotopes is extremely scalable for affine dynamics [3, 77]. The approach has been extended to nonlinear differential algebraic equations [2].

Support Functions. A support function represents a closed, bounded, and convex set exactly, somewhat like a characteristic function. Support functions lead to very scalable algorithms since linear map, Minkowski sum, and convex hull correspond to simple operations on vectors and scalars [74, 83, 116].

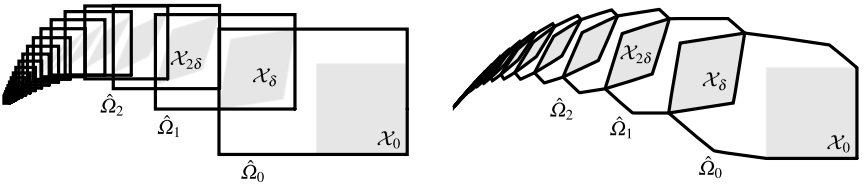
The *support function* $\varrho_P : \mathbb{R}^n \to \mathbb{R}$ of a nonempty, closed, bounded, and convex set P is

$$\varrho_P(d) = \max\{d^\mathsf{T} x \mid x \in P\}.$$

It attributes to every *direction* $d \in \mathbb{R}^n$ the position of the tangent halfspace in that direction, see Fig. 10(a). The values of the support function over a set of directions $D \subseteq \mathbb{R}^n$ define an *outer approximation*

$$\lceil P \rceil_D = \bigcap_{d \in D} \{d^\mathsf{T} x \leq \varrho_P(d)\}.$$

If $D = \mathbb{R}^n$ or D is the ball of a norm, then $\lceil P \rceil_D = P$, which shows that the support function indeed represents the set exactly. If D is a finite set of directions, the outer approximation is a polyhedron, as illustrated in Fig. 10(b) and applied to reachability in Fig. 11. While for a given direction the numerical value of the support function



(a) using the axis directions gives a bounding box approximation of Ω_k from (14)

(b) adding more directions, the approximation approaches Ω_k from (14)

Fig. 11 A reach set cover can be computed with support functions and initial approximations Ω_0, Ψ_δ from a variation of Lemma 3 where the bloating is non-uniform [68]. Evaluating the support function in a given set of directions results in the shown outer approximation $\hat{\Omega}_0, \hat{\Omega}_1, \ldots$ (*solid*)

can often be computed very efficiently, one does not escape the curse of dimensionality if the goal is to compute an outer approximation of a given accuracy: To obtain an outer approximation within a Hausdorff distance ε of P in n dimensions, one needs to evaluate the support function in $\mathcal{O}(\frac{1}{\varepsilon^{n-1}})$ directions. Asymptotically optimal algorithms to construct ε-close approximations are described, e.g., in [116]. However, for some examples even a small number of directions can lead to reachability results with an acceptable approximation error [68].

Linear map, Minkowski sum, and convex hull are easily computed with support functions:

$$\begin{aligned}\varrho_{AP}(d) &= \varrho_P\big(A^{\mathsf{T}}d\big),\\ \varrho_{P\oplus Q}(d) &= \varrho_P(d) + \varrho_Q(d),\\ \varrho_{\mathrm{chull}(P\cup Q)}(d) &= \max\big\{\varrho_P(d), \varrho_Q(d)\big\}.\end{aligned}$$

The intersection operation is more complex, and can be formulated as an optimization problem [83].

Thanks to the above properties, support functions serve well as a lazy representation for sets that arise from the successor computations described in Sect. 30.4.3.1. Computing the support function of the sequence (14) for a given direction can be done very efficiently even without the approximation operator Appr [83].

Two issues need to be solved to use support functions efficiently in the reachability Algorithm 1. First, the nesting of support functions should be of limited depth, in particular because evaluating the support function of an intersection operation requires multiple evaluations of its operands. Second, deciding containment is hard for support functions. Both problems can be solved by switching the set representation from a support function to its polyhedral outer approximation at appropriate points in the algorithm [68]. Combining support functions and polyhedral computations for a fixed set of directions D is closely related to reachability with *template polyhedra* [160] and both require that a good set of directions D be chosen. The support function representation can be extended to represent the entire (non-convex) reachable set by parameterizing it over time [69].

Polyhedra. The class of polyhedra is closed under all required operations, i.e., linear map, Minkowski sum, convex hull, and intersection. However, not all of them scale well. As mentioned in Sect. 30.4.2, there are no scalable algorithms for computing convex hull and Minkowski sum on polyhedra in constraint representation. For illustration, consider that using the convex hull of n line segments, each given by $2n$ constraints in n dimensions, one can construct a cross-polytope, which has 2^n constraints. Taking the Minkowski sum can lead to a similar explosion in the number of constraints. This is illustrated by the fact that the Minkowski sum can be computed with a convex hull and an intersection operation in $n+1$ dimensions using the *Cayley Trick* [173]. A polyhedral approximation for the non-scalable operations can be efficiently computed by a priori fixing the facet normals of the result, e.g., using the outer approximation of the support function. The accuracy of the approximation can be increased by including additional directions, leading to a scalable approach [20].

30.4.3.3 Clustering

The accuracy of the approximation in Lemma 3 depends on the size of the time step. This property, common to all approaches cited in Sect. 30.4.3, points to a potential bottleneck: To achieve a desired accuracy, one may end up with a large number of sets to cover the required time horizon. In the next successor computation, each one of these sets may become the initial set of yet another sequence, and so one may easily end up with an exponential increase in the number of sets. If only very few of these sets intersect with the guard sets, the discrete successor computation results in few sets and therefore acts as a filter that might just keep the number of sets manageable. But this is not the case in general; note that these sets necessarily overlap. To prevent an explosion in the number of sets, a common approach is to cluster together all sets that intersect with the same guard [83]. The clustering operation, e.g., taking the convex hull, can itself be costly and adds to the approximation error in a way that is not easy to quantify. An approach to obtain a suboptimal number of clusters for a given error bound is presented in [69].

30.4.4 Nonlinear Dynamics

We give a very brief overview of techniques that deal with nonlinear dynamics

$$\dot{x} = f(x),$$

where f is usually assumed to be globally Lipschitz continuous.

Linearization. One way to deal with nonlinear dynamics is to approximate them with affine dynamics $\dot{x} = Ax + u, u \in \mathcal{U}$ and then use reachability algorithms for

affine dynamics. First, the states are confined to a bounded domain $\mathcal{S}$. This can be the evolution domain restriction in a location, or $\mathcal{S}$ can be derived iteratively by growing suitable bounds around a given set of initial states. Then, a suitable matrix A and vector b are chosen. For example, linearizing $f(x)$ around a point $x_0 \in \mathcal{S}$ gives a matrix A with elements $a_{ij} = \frac{\partial f_i}{\partial x_j}|_{x=x_0}$ and a vector $b = f(x_0) - Ax_0$. Finally, one derives a set $\mathcal{U}_\varepsilon$ that bounds the error such that for all $x \in \mathcal{S}$,

$$f(x) - (Ax + b) \in \mathcal{U}_\varepsilon.$$

Such bounds can be obtained using, e.g., interval arithmetic or optimization techniques. The states reachable using the affine dynamics $\dot{x} = Ax + u,\ u \in \mathcal{U}_\varepsilon \oplus \{b\}$ cover those of the original nonlinear dynamics. This approach constructs an *abstraction* of the system. Such abstractions are discussed more formally in Sect. 30.5.2.

The accuracy of the linearization depends on the size of the domain $\mathcal{S}$. It can be increased by partitioning $\mathcal{S}$ into smaller parts. Each part can then be associated with smaller error bounds $\mathcal{U}_\varepsilon$ and consequently gives a more accurate approximation of the reachable set. The switching of the system from one element of the partition to another is straightforward to model with a hybrid automaton. This process is known as *phase-portrait approximation*, see also Sect. 30.5.2. It can be of use even when dealing with purely continuous dynamical systems, in which case it is also referred to as *hybridization* [18]. The abstract model can be simplified by projecting away variables and adding a clock variable to preserve timing properties [17].

Polynomial Approximations. If the dynamics are polynomial, bringing them into Bernstein form allows one to compute conservative approximations of successor sets in polynomial form [54, 152]. Another approach is to use *Taylor models*, which are polynomial approximations of a function that are derived from a higher-order Taylor expansion and an interval bound on the remainder [30]. The resulting ODE can be solved by iterative approximations using the Picard operator. The reachable states are approximated by sets that are polyhedra [160] or polynomial images of intervals [43]. A similar approach uses polynomial images of zonotopes, which are themselves images of intervals [4]. Since polynomial images of intervals are generally not closed under intersection, the accuracy may be diminished when computing discrete successors. It can also be shown that additional assumptions, such as knowledge of a Lipschitz constant, are required in these approaches in order to ensure computable error bounds [147].

30.5 Abstraction-Based Verification

Explicit-state reachability analysis is very easy to use. Its flat and direct representation of the system behavior can, however, cause it to run into scalability issues for bigger systems. One technique that has been very successful for scaling up discrete model checking is that of abstraction (see also Chap. 13).

The basic idea is to replace the actual system by a simpler, abstract system, in which model checking is easier to perform. The verification results about the abstract system, of course, can only be related back to verification results about the original concrete system under certain conditions on how the abstract and concrete system are related and whether the particular property in question survives this abstraction process.

The options for directly constructing discrete abstractions by finite quotients and for which subclasses they work have been examined by Henzinger [88, 93] and Lafferriere et al. [108]. Because of the limited scope of discrete abstractions, more general predicate abstractions [9, 10] and abstraction refinement techniques like Counterexample-Guided Abstraction Refinement (CEGAR) have been developed subsequently [9, 46]; see Chap. 13. These directions have again worked successfully in discrete and, to some extent, real-time systems.

30.5.1 Discrete Abstractions

We present a general notion of abstraction for transition systems based on simulation relations [125] and we illustrate the principle of using abstractions in the verification of hybrid systems for the class of initialized rectangular automata.

Definition 11 (Abstraction) A transition system $T^A = \langle S^A, S_0^A, S_f^A, \Sigma, \rightarrow_A \rangle$ is an *abstraction* of a transition system (with the same alphabet) $T = \langle S, S_0, S_f, \Sigma, \rightarrow \rangle$ (which is then called the *concrete* system) if there exists an *abstraction mapping* $\alpha : S \rightarrow S^A$ such that the following conditions hold:

1. $\alpha(s) \in S_0^A$ for all initial states $s \in S_0$;
2. for all $\sigma \in \Sigma$, for all states $s_1, s_2 \in S$, if $s_1 \xrightarrow{\sigma} s_2$, then $\alpha(s_1) \xrightarrow{\sigma}_A \alpha(s_2)$;
3. $\alpha(s) \in S_f^A$ for all final states $s \in S_f$.

The abstraction mapping α is in fact a particular case of a (time-abstract) simulation relation [126]. It may be convenient to allow the abstraction mapping to map a state $s \in S$ to several abstract states $s_A^1, s_A^2, \ldots, s_A^k \in S^A$, that is to consider abstraction mappings $\alpha : S \rightarrow 2^{S^A}$ or equivalently to consider an abstraction relation over $S \times S^A$, rather than a function. We take the simpler definition which is sufficient for the purpose of describing the main principles of abstraction for hybrid automata.

The main property of abstractions which is useful for the safety verification problem of hybrid automata is that they are conservative. Formally, $\{\alpha(s) \mid s \in \mathsf{Reach}(T)\} \subseteq \mathsf{Reach}(T^A)$, which implies the following.

Lemma 4 *Let T^A be an abstraction of T. If T^A is safe, then T is safe.*

By Lemma 4, if we show (e.g., using algorithmic techniques) that the unsafe states are not reachable in an abstraction of a hybrid system, then we can conclude

that the concrete system is safe. Intuitively, this is because abstractions are over-approximations of the original system, and therefore they exhibit (or simulate) all executions of the concrete system, and possibly more. In particular, every path to an unsafe state has a matching path in the abstraction, which is the main argument for proving Lemma 4. The converse of this lemma does not hold simply because abstractions may introduce spurious executions (which have no matching execution in the concrete system) due to over-approximation.

The main purpose of abstraction for hybrid systems is to obtain finite-state transition systems which are amenable to model checking by automated tools, and give useful conclusions about the original system. Remember that the transition systems of hybrid automata have (uncountably) infinite state space, and (uncountably) infinite branching. In the next subsections, we present ideas for practically constructing such abstractions.

Initialized Rectangular Automata. We illustrate abstractions with an informal argument of why the safety verification problem is decidable for initialized rectangular automata. The idea is that for such hybrid automata H, one can construct a timed automaton A such that A is an abstraction of H, and H is an abstraction of A, thus A is safe if and only if H is safe. Note that in this case the constructed abstraction (the timed automaton A) has infinite state space, but since we know that the safety verification problem for timed automata is decidable, we obtain decidability for initialized rectangular hybrid automata by Lemma 4.

We present the main steps behind this construction. In every location, a variable x with flow constraint $k_1 \leq \dot{x} \leq k_2$ is replaced by two variables x_l and x_u with flow constraint $\dot{x}_l = k_1$ and $\dot{x}_u = k_2$ which track the least and greatest possible value of x respectively. An incoming edge with jump condition $a \leq x^+ \leq b$ (an update) is replaced by $x_l^+ = a \wedge x_u^+ = b$. An edge with jump condition $x \leq b$ (a guard) that occurs in conjunction with $x^+ = x$ is replaced by two copies of the edge, one with the constraint $(x_l \leq b \wedge x_u \geq b \wedge x_u^+ = b)$ and the other with the constraint $x_u \leq b$. More complicated jump conditions (strict inequalities, and conjunction of simple jump conditions) are handled analogously, as well as the constraints in initial, final, and evolution domain conditions (invariants).

After this step, the slope of every variable is a singleton in every location. The next step is to scale the nonzero slope of the variables to 1. To do this, in each location we replace flow constraints $\dot{x} = k$ (when $k \neq 0$) by $\dot{x} = 1$ and divide by k the constants in the guards of outgoing edges, and in the updates of incoming edges. This ensures that the value stored in variable x remains k times smaller than the value of x in the original automaton (as long as the flow constraint $\dot{x} = k$ holds). It is therefore important that the rectangular automaton is initialized, as it guarantees that if the constraint $x^+ = x$ occurs in the jump condition of an edge (ℓ, σ, ℓ'), then the slope of variable x is the same in ℓ and in ℓ'. It remains to eliminate variables with slope 0, which can be done easily by storing the lower and upper value of x in the finite control structure of the automaton (these values can be changed only by discrete jumps).

The technical details of how to deal for instance with strict constraints in a jump condition like ($a < x < b$), or unbounded flow constraints (like $\dot{x} \geq 1$) can be found in [96].

30.5.2 Phase-Portrait Approximation

Phase-portrait approximations are used as abstractions of hybrid automata with complex flow constraints. We discuss the approach for affine flow constraints, but it also applies to flow constraints that are much more general (e.g., given by $\dot{x} = f(x)$ for a continuous function f). Details about the theory and practice of this approach can be found, e.g., in [67, 93] and extensions on hybridization in [18] and other abstractions in [60, 70, 169].

The objective of phase-portrait approximations is to replace complex dynamics by simple rectangular (or sometimes linear) flow constraints on the dotted variables only. For example, the flow constraint $\dot{x} = f(x)$ where $f(x) = x + 1$ in a location with evolution domain (invariant) $0 \leq x \leq 10$ is replaced by $1 \leq \dot{x} \leq 11$, which over-approximates the exact dynamics. In general, bounds on the derivative can be derived from bounds on the variables and computed as optimization problems, where the lower bound should be less than of equal to $\inf_{v \in [\![\mathsf{Inv}(\ell)]\!]} f(v)$ and symmetrically for the upper bound. Manual or numerical methods can be used as long as the bounds can be proven to hold.

Formally, a *phase-portrait approximation* of a hybrid automaton $H = \langle \mathsf{Loc}, \mathsf{Lab}, \mathsf{Edg}, X, \mathsf{Init}, \mathsf{Inv}, \mathsf{Flow}, \mathsf{Jump}, \mathsf{Final} \rangle$ is a hybrid automaton $H' = \langle \mathsf{Loc}, \mathsf{Lab}, \mathsf{Edg}, X, \mathsf{Init}, \mathsf{Inv}, \mathsf{Flow}', \mathsf{Jump}, \mathsf{Final} \rangle$ in which all components in H and H' are identical, except the flow constraint which is such that $[\![\mathsf{Flow}'(\ell)]\!] \supseteq [\![\mathsf{Flow}(\ell)]\!]$ for every location $\ell \in \mathsf{Loc}$.

Lemma 5 *Let H' be a phase-portrait approximation of H. Then $[\![H']\!]$ is an abstraction of $[\![H]\!]$, and if $[\![H']\!]$ is safe, then $[\![H]\!]$ is safe.*

The safety verification problem for phase-portrait approximations can be solved using the algorithms and data structure presented in Sect. 30.4 for reachability analysis. Rectangular phase-portrait approximations are relatively simple to obtain because bounds are computed for each variable separately. However, the quality of the approximation may be too coarse to establish safety. If the bad states are reachable in the phase-portrait approximation, it may be due to lack of accuracy. More precise approximations are obtained by splitting the evolution domains. For example, a location with evolution domain $0 \leq x \leq 10$ can be replaced by two locations with respective evolution domains $0 \leq x \leq 5$ and $5 \leq x \leq 10$, over which the approximation of $\dot{x} = x + 1$ is more precise, namely $1 \leq \dot{x} \leq 6$ and $6 \leq \dot{x} \leq 11$ respectively. Figure 12 shows the states reachable from $x = t = 0$ (assuming $\dot{t} = 1$) in the rectangular phase-portrait approximation before splitting (light gray) and after splitting (dark gray).

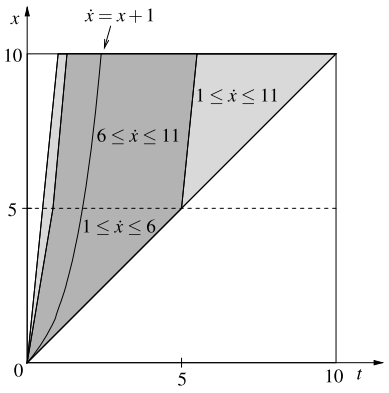


Fig. 12 Tighter approximations using evolution domain (invariant) splitting

In general, splitting consists of replacing a location ℓ by k locations $\ell_1, \ldots, \ell_k$ with the same flow constraint as in ℓ, and with evolution domains that cover the evolution domain of ℓ, i.e., such that $[\![\mathsf{Inv}(\ell)]\!] \subseteq \bigcup_{i=1}^{k} [\![\mathsf{Inv}(\ell_i)]\!]$. For each incoming edge (ℓ', σ, ℓ), new edges (ℓ', σ, ℓ_i) $(i = 1, \ldots, k)$ are created with the same jump condition, and similarly for each outgoing edge. The split locations $\ell_1, \ldots, \ell_k$ are connected by edges with jump condition $\mathit{stable}(X) = \bigwedge_{x \in X} x' = x$. It can be shown that location splitting results in hybrid automata that are mutually abstractions of each other, implying that one is safe if and only if the other is safe. By splitting locations, rectangular phase-portrait approximation can be made arbitrarily precise in the following sense. Given a hybrid automaton H and $\varepsilon > 0$, an ε-relaxation of H is a hybrid automaton with the same locations and transition structure as in H, and where all predicates ϕ in H are replaced by predicates ϕ' such that $[\![\phi]\!] \subseteq [\![\phi']\!] \subseteq [\![\phi]\!]_\varepsilon$ where $[\![\phi]\!]_\varepsilon := \{v \in \mathbb{R}^X \mid \exists u \in [\![\phi]\!] : \max_{x \in X} |v(x) - u(x)| \leq \varepsilon\}$ is the set of valuations at a distance at most ε from a valuation satisfying ϕ.

It can be shown that for every hybrid automaton H and $\varepsilon > 0$, there exists a rectangular phase-portrait approximation H_ε of a splitting of H such that H_ε is an abstraction of H, and there exists an ε-relaxation of H which is an abstraction of H_ε (see [93]). This ensures that if H robustly satisfies a safety property (i.e., both H and some ε-relaxation satisfy the safety property), then it is possible to establish the property using rectangular phase-portrait approximation and splitting.

In practice, it is often useful to split locations according to specific information we may have about the given hybrid automaton. For example, a flow constraint $\dot{x} = 3 - x$ suggests the evolution domain should be split along lines parallel to $L \equiv 3 - x = 0$. More generally, a common heuristic is to use linear approximations of the flow constraints as support for cutting planes.

30.5.3 Predicate Abstractions

This part is a survey of [9, 10, 46]. We provide general ideas and guidelines about predicate abstraction schemes for hybrid systems. Chapter 13 provides a detailed

presentation of abstraction techniques for program verification (note that imperative programs can be viewed as a subclass of hybrid systems).

Reachability analysis based on predicate abstraction consists of tracking the truth value of a fixed finite set of predicates instead of computing the value of the continuous variables. The continuous part of the state space is replaced by the Boolean truth values of the predicates.

Let H be a hybrid system, and let $\Pi = \{\pi_1, \ldots, \pi_k\}$ be a finite set of linear predicates π_i of the form $y \bowtie 0$ where $y \in \mathsf{LTerm}(X)$ and $\bowtie \in \{<, \leq, =, >, \geq\}$. A truth value for Π is a vector $b \in \mathbb{B}^k$ where $\mathbb{B} = \{0, 1\}$ that assigns a truth value b_i to each predicate $\pi_i \in \Pi$. Truth values induce a partition of the continuous state space into finitely many abstract states. To obtain an abstraction we require that whenever there exists a transition between two concrete states, then there is a transition between the corresponding abstract states. Hence, the transition relation satisfies Definition 11 by construction.

We define an abstraction mapping α_Π as follows. For all states (ℓ, v) of the hybrid automaton H, let $\alpha_\Pi(\ell, v) = (\ell, b)$ if $b = (b_1, \ldots, b_k) \in \mathbb{B}^k$ is the vector of truth values of the predicates in Π under valuation v, i.e., such that $\pi_i(v) = b_i$ for all $1 \leq i \leq k$. We sometimes omit the location and write $\alpha_\Pi(v) = b$. We denote by γ_Π the *concretization function* such that $\gamma_\Pi(b) = \{v \in \mathbb{R}^X \mid \alpha_\Pi(v) = b\}$ for all $b \in \mathbb{B}^k$.

The *predicate abstraction* of H induced by Π is the finite-state transition system $H_\Pi = \langle S_\Pi, S_0, S_f, \Sigma, \rightarrow_\Pi \rangle$ where:

- $S = \{(\ell, b) \in \mathsf{Loc} \times \mathbb{B}^k \mid \exists v \in [\![\mathsf{Inv}(\ell)]\!] : \alpha_\Pi(\ell, v) = (\ell, b)\}$
- $S_0 = \{(\ell, b_0) \in S_\Pi \mid \exists v \in [\![\mathsf{Init}(\ell)]\!] : \alpha_\Pi(\ell, v) = (\ell, b_0)\}$;
- $S_f = \{(\ell, b_f) \in S_\Pi \mid \exists v \in [\![\mathsf{Final}(\ell)]\!] : \alpha_\Pi(\ell, v) = (\ell, b_f)\}$;
- $\Sigma = \mathsf{Lab} \cup \{\mathsf{time}\}$ where Lab is the alphabet of H;
- For each $\sigma \in \mathsf{Lab}$, the transition relation $\rightarrow_\Pi$ contains all tuples $((\ell, b), \sigma, (\ell', b'))$ such that $\exists e = (\ell, \sigma, \ell') \in \mathsf{Edg} \cdot \exists v \in \gamma_\Pi(b) \cdot \exists v' \in \gamma_\Pi(b') : (\ell, v) \xrightarrow{\sigma} (\ell', v')$; and the transition relation $\rightarrow_\Pi$ contains the tuples $((\ell, b), \mathsf{time}, (\ell', b'))$ such that $\ell' = \ell$ and $\exists r \geq 0 \cdot \exists v \in \gamma_\Pi(b) \cdot \exists v' \in \gamma_\Pi(b') : (\ell, v) \xrightarrow{r} (\ell, v')$.

While predicate abstractions are finite-state, their size can be of prohibitive computational cost. The number of states in H_Π is at most exponential in the number of predicates in Π. In practice though, many truth value vectors are not feasible (i.e., they have an empty concretization). For example, think of a set of $2k$ predicates over two variables x and y, where k predicates define a partition of the values for x (e.g., $x < 0$, $0 \leq x \leq 1$, and $1 < x$) and k predicates define a partition of the values for y. Then the number of feasible abstract states is at most k^2 rather than 2^{2k}. Note that this example would still give a number of abstract states exponential in the number of variables. The dimension of the space is a well-known source of computational complexity. The choice of predicates is thus very important to obtain precise approximations at the least cost. The initial set of predicates is usually chosen manually. Natural candidates are the predicates occurring in the hybrid automaton itself, like the evolution domains and jump conditions. Automatic construction and refinement of predicate abstractions is discussed in Sect. 30.5.4.

For reachability analysis, it is usually not necessary to construct the entire transition systems of the predicate abstractions, because many states may not be reachable. On-the-fly approaches are used to simultaneously construct and explore the abstraction. Starting from the initial states in the abstraction, the transitions to other abstract states are explored as and when they are computed. A classical strategy is to explore the discrete successors first (because they are less expensive to compute), and then the continuous successors for increasing amounts of time, as long as no new discrete transition is enabled.

Computing Discrete Successors. Discrete successors can be computed as follows. Given $b \in \mathbb{B}^k$, let $\Pi(b) = \bigwedge_{i|b_i=1} \pi_i \wedge \bigwedge_{i|b_i=0} \overline{\pi_i}$ be the constraint defining the abstract state b. A transition $e = (\ell, \sigma, \ell')$ is *enabled* in a state (ℓ, b) if $\mathsf{EN}(e) := [\![\exists X \cdot \Pi(b) \wedge \mathsf{Inv}(\ell) \wedge \mathsf{Jump}(e)]\!] \cap [\![\mathsf{Inv}(\ell')]\!] \neq \emptyset$. The successor states of (ℓ, b) by enabled transition e are the abstract states (ℓ', b') such that $R_k \neq \emptyset$ where $R_0 = \mathsf{EN}(e)$ and for all $1 \leq i \leq k$, if $b'_i = 1$ then $R_i = R_{i-1} \cap [\![\pi_i]\!]$, and if $b'_i = 0$ then $R_i = R_{i-1} \cap [\![\overline{\pi_i}]\!]$. A procedure for computing b' can easily be derived from this definition. Note that it may be that both $R_{i-1} \cap [\![\pi_i]\!] \neq \emptyset$ and $R_{i-1} \cap [\![\overline{\pi_i}]\!] \neq \emptyset$ hold, which would lead $b'_i = 1$ and $b'_i = 0$ to be set successively, and both cases to be explored. A simple optimization of this procedure is for each $1 \leq i \leq k$ to set $b'_i = 1$ beforehand if $\mathsf{EN}(e) \cap [\![\overline{\pi_i}]\!] = \emptyset$, and set $b'_i = 0$ if $\mathsf{EN}(e) \cap [\![\pi_i]\!] = \emptyset$. If one of the two cases holds, then the corresponding predicate can be skipped in the computation of R_i's.

Computing Continuous Successors. In general, the continuous successors are not computed exactly, even according to the abstract transition relation. This is due to the lack of exact algorithmic methods for solving differential equations. Note that this is a difficult problem even if the differential equations in the flow constraints have closed-form solutions, like in linear systems. Given $R \subseteq \mathbb{R}^X$ and location ℓ, we want to compute the set $\mathsf{Post}_C(\{\ell\} \times R)$ of continuous successor states as defined in Sect. 30.4.1, but over-approximations are sufficient for our purpose. This is consistent with the framework of abstraction (in the sense of Definition 11), but strictly speaking we are exploring in this way an over-approximation of the transition system H_Π defined above.

Optimizations. Various optimizations and heuristics have been defined and evaluated on many examples in the literature, see, e.g., [9, 10, 46]. For example, when we discover that a new abstract state s is reachable as a continuous successor under some flow constraint, we do not need to explore the continuous successors of s under the same flow constraint (unless s is also reachable by some discrete transition). This may significantly prune the search through the abstract state space. The search can also be guided to discover unsafe reachable states as quickly as possible. Various exploration strategies have been defined, based on giving priority to the most promising states, according to some greedy measures. For example, such measures may estimate the distance from the current state to the unsafe state, such as the Euclidean distance between the valuation of the variables in the abstract state

and in the unsafe states, or a discrete distance as the smallest number of discrete transitions necessary to reach an unsafe state, possibly taking into account the jump condition on the edges. Combination thereof are also possible [10]. Finally, as in program verification [90], it may be useful to maintain a set of predicates Π specific to each location, because certain predicates that are relevant in one location may not be useful in other locations.

30.5.4 Abstraction Refinement

The abstraction schemes presented in Sect. 30.5.1 and Sect. 30.5.3 may not be sufficient to establish the safety of a system. In particular, we know that safety of the abstraction implies safety of the original system, but non-safety of the abstraction is inconclusive. The process of refinement consists of constructing abstractions that are tighter (or more detailed) than a given abstraction, in order to prove safety. If the refinement process repeatedly fails in proving safety, then one can reasonably conclude that even if the original system may indeed be safe, it should not be considered as acceptable because its correctness is not robust, a small deviation in the implementation of the system being able to cause violation of the safety requirement [58, 155]. Such considerations are used to stop the refinement process when a specified level of precision is reached [46, 64].

In general, if T^A is an abstraction of T, then a *refinement* T^B of T^A is an abstraction of T which is such that T^A is an abstraction of T^B.

In the case of splitting and phase-portrait approximations, refinements can be obtained by further splitting locations. For predicate abstractions, adding new predicates gives a refinement. We present one of the most popular frameworks to discover new predicates automatically, the counterexample-guided abstraction refinement (CEGAR) [46, 47]. A general framework of abstraction refinement is presented in Chap. 13.

Spurious Counterexamples. When a predicate abstraction fails to establish safety, the analysis usually returns a witness path from an initial abstract state to a final abstract state. Such a path $\rho = q_0 \xrightarrow{\sigma_1} q_1 \dots \xrightarrow{\sigma_n} q_n$ is a *spurious* counterexample if there exists no path $(\ell_0, v_0) \xrightarrow{\sigma_1} (\ell_1, v_1) \dots \xrightarrow{\sigma_n} (\ell_n, v_n)$ in the original system such that $(\ell_i, v_i) \in \gamma_\Pi(q_i)$ for all $0 \leq i \leq n$. Clearly, if a counterexample is not spurious, then we can immediately conclude that the original system is not safe. We present a standard approach to check whether a counterexample is spurious [9].

To simplify the presentation, we assume in this section that every edge has a different label that identifies it uniquely. The successor operator is

$$\mathsf{Post}_\sigma(S) = \left\{ \left(\ell', v'\right) \mid \exists (\ell, v) \in S : (\ell, v) \xrightarrow{\sigma} \left(\ell', v'\right) \right\},$$

where $\mathsf{Post}_{\mathsf{time}}(\cdot) = \mathsf{Post}_C(\cdot)$ is the one-step continuous successor operator. Similarly, the predecessor operator is

$$\mathsf{Pre}_\sigma(S) = \left\{(\ell, v) \;\middle|\; \exists\left(\ell', v'\right) \in S : (\ell, v) \xrightarrow{\sigma} \left(\ell', v'\right)\right\}.$$

Let $R_0 = \gamma_\Pi(q_0) \cap \{(\ell, v) \mid v \in [\![\mathsf{Inv}(\ell)]\!]\}$, and $R_{i+1} = \mathsf{Post}_{\sigma_i}(R_i) \cap \gamma_\Pi(q_{i+1}) \cap \{(\ell, v) \mid v \in [\![\mathsf{Inv}(\ell)]\!]\}$ for all $i \geq 0$. The counterexample ρ is spurious iff $R_i = \emptyset$ for some $0 \leq i \leq n$. Note that over-approximations of $\mathsf{Post}_{\sigma_i}(\cdot)$ may suffice to show that a counterexample is spurious, but under-approximations are necessary to establish with certainty that a counterexample exists in the original system.

Refinement. Assume that the counterexample is spurious, and let $j \geq 0$ such that $R_j \neq \emptyset$ and $R_{j+1} = \emptyset$. Then it is easy to prove that $R_j \cap \mathsf{Pre}_{\sigma_{j+1}}(\gamma_\Pi(q_{j+1})) = \emptyset$. New predicates should be added to the set Π in order to rule out the counterexample. Since $R_j \cap \mathsf{Pre}_{\sigma_{j+1}}(\gamma_\Pi(q_{j+1})) = \emptyset$, we can search for a set of predicates which separates R_j and $\mathsf{Pre}_{\sigma_{j+1}}(\gamma_\Pi(q_{j+1}))$. A set $\hat{\Pi}$ of predicates *separates* two sets R and Q if for every truth value $b \in \mathbb{B}^{\hat{\Pi}}$, we have either $\gamma_{\hat{\Pi}}(b) \cap R = \emptyset$ or $\gamma_{\hat{\Pi}}(b) \cap Q = \emptyset$.

Note that to separate closed polyhedra, one simple linear constraint is always sufficient, but since reachable states (and in particular states reachable by continuous flow) are approximated by non-convex unions of polyhedra, several simple constraints may be necessary. Several methods have been developed to separate polyhedral sets, which are beyond the scope of this chapter. We refer to [9] for references and discussion.

In some case, spuriousness can be established by analyzing fragments of the counterexample [46], i.e., trying to show that a sub-sequence in the counterexample is not feasible in the original system. Spurious fragments of length 2 are called *locally infeasible* in [9] and defined as follows: $q_{i-1} \xrightarrow{\sigma_i} q_i \xrightarrow{\sigma_{i+1}} q_{i+1}$ is spurious if $\mathsf{Post}_{\sigma_i}(\gamma_\Pi(q_{i-1})) \cap \gamma_\Pi(q_i) \cap \mathsf{Pre}_{\sigma_{i+1}}(\gamma_\Pi(q_{i+1})) = \emptyset$. Refinement is computed as above using separating predicates.

Various forms of robustness have been considered for hybrid systems, which basically work by not distinguishing between almost safe and almost unsafe hybrid systems so that incorrect answers from the analysis procedure are accepted for such borderline cases, but correct answers are required for clear-cut cases. Different notions of robustness have been considered successfully [64, 157, 158].

30.5.5 Approximate Bisimulations

For discrete systems, the relationships between systems can be described by the notions of language inclusion, simulation, and bisimulation. These concepts have been transposed to continuous and hybrid systems [85], and extended to take advantage of metrics over state spaces [78]. While traditional simulation and bisimulation relations require the output traces of related states to be identical, it suffices for metric

relations that they are sufficiently close. It is then possible to construct a discrete bisimilar quotient by discretizing the state space. The quotient is then amenable to verification and controller synthesis techniques for discrete systems [76].

We briefly sketch out the principle of approximate bisimulations in discrete time. Two states x_1, x_2 are in *ε-bisimulation relation* if their output values are within distance ε and for every successor state x_1' of x_1, x_2 has a matching successor state x_2' so that x_1' and x_2' are also in the relation. As a consequence, the output traces of two states in the relation will never be more than ε apart. Note that the definition coincides with classical bisimulation for $\varepsilon = 0$. It is generally hard to compute ε-bisimulation relations exactly, but (under some mild assumptions) one can define a Lyapunov-like *bisimulation function* that maps pairs of states to a non-negative value, and whose sub-level sets are in an approximate bisimulation relation. The existence of bisimulation relations can be tied to certain types of stability (the tendency of the system to go to its equilibrium point). For example, a bisimulation function of a linear continuous system with dynamics $\dot{x} = Ax$ and output signal $y = Cx$ can be computed efficiently even for high-dimensional systems by solving a set of linear matrix inequalities (LMI) of the form $M \geq C^{\mathsf{T}}C$ and $A^{\mathsf{T}}M + MA \leq 0$. The LMI always has a solution if the system is stable. Therefore, two stable linear systems are always ε-bisimilar, and an upper bound on ε can be computed. Note that approximate bisimulations can be used to relate continuous-time to discrete-time systems, continuous-valued to discrete-valued systems, etc.

Verification by Simulation. Bisimulation relations can also be used to verify bounded-horizon properties on bounded regions by computing a finite number of trajectories, a technique called *verification by simulation* [61, 102]. Here, the proximity measure of the bisimulation relation is combined with a robustness measure on temporal logic formulas. Given an initial state x_0 from which a trajectory satisfies a temporal formula to some measure, a bisimulation metric allows one to identify a neighborhood of initial states that all satisfy the same formula. This is possible since the bisimulation metric guarantees that all trajectories from the neighborhood (including all trajectories starting in x_0) remain sufficiently close together to satisfy the formula. Given a (dense) bounded region of initial states, it is, under suitable assumptions, possible to identify a finite subset of initial states whose trajectories are sufficient to show that the system satisfies a temporal logic formula [79]. A similar approach has been developed for embedded control software [112]. Together with the work on robustness mentioned in Sect. 30.5.4, these results demonstrate how stability and robustness can be used to simplify verification tasks.

30.6 Logic-Based Verification

The working principle behind logic-based verification is to use logical formulas to characterizing some parts of the hybrid systems verification problem and to solve this verification problem or subproblem entirely by checking the corresponding logical formulas for validity. There are even verification techniques for hybrid systems

that are entirely based on logic and proof [137, 143], which are beyond the scope of this chapter, however. In this section we survey the basic principles behind these approaches and show what kind of reasoning can be used to verify safety properties of hybrid systems or their parts by showing the validity of logical formulas.[7]

We survey a number of different approaches that represent the verification problem by various logical formulas or logical constraints:

1. Polynomial barrier certificates [154];
2. Equational certificates from templates [159, 161];
3. Differential invariants [136, 142, 144, 148].

These logic-based verification approaches further have in common that they argue by invariance and are based on variations of the work of Sophus Lie, of Jean Gaston Darboux, or of Aleksandr Lyapunov. Differential invariants are based on Sophus Lie's 1867–1873 work on what are now called Lie derivatives and Lie groups. Equational certificates are based on Darboux's 1878 results [56] on a way to use Sophus Lie's approach. Barrier certificates are based on variations of Aleksandr Lyapunov's 1884–1892 work on a criterion for stability, which is used for safety instead [154]. The logic-based verification techniques for hybrid systems are complementary, so barrier certificates, equational templates, and differential invariants can be used together and also combined as abstractions with reachability analysis techniques.

Consider a location ℓ of a hybrid automaton with polynomial dynamics defined by polynomial differential equations. To emphasize that such a differential equation is considered only once even if it occurs in multiple different locations, it is also referred to as *continuous mode*. Let

$$\dot{x_1} = f_1(x), \ldots, \dot{x_n} = f_n(x)$$

be the polynomial differential equation system of the mode, which we abbreviate by the (vectorial) differential equation $\dot{x} = f(x)$. The mode ℓ has an invariant condition $\mathsf{Inv} \in \mathsf{PConstr}(X)$. What we want to understand in model checking of safety properties is whether the system will always stay in a safe region when it follows this continuous evolution mode starting from some initial region. We represent the desired initial region by a constraint $\mathsf{Init} \in \mathsf{PConstr}(X)$. Finally, we consider a constraint $\mathsf{Safe} \in \mathsf{PConstr}(X)$ defining the safe states for which we want to show that our system never leaves the set of states $[\![\mathsf{Safe}]\!]$ satisfying Safe.

Definition 12 (Continuous mode safety problem) Let $\dot{x} = f(x)$ be a (vectorial) differential equation, i.e., a polynomial differential equation system

$$\dot{x_1} = f_1(x), \ldots, \dot{x_n} = f_n(x)$$

[7] It should be noted that the other verification techniques surveyed in this chapter benefit from logic as well, for example in their representation of big sets of states using simple logical formulas.

for the system variables $X = \{x_1, \ldots, x_n\}$. A *continuous system* $(\mathsf{Init}, \dot{x} = f(x), \mathsf{Inv})$ consists of a constraint $\mathsf{Inv} \in \mathsf{PConstr}(X)$ for the invariant condition (or evolution domain restriction), and a constraint $\mathsf{Init} \in \mathsf{PConstr}(X)$ for the initial condition. We say that the continuous system $(\mathsf{Init}, \dot{x} = f(x), \mathsf{Inv})$ is *safe with respect to* constraint $\mathsf{Safe} \in \mathsf{PConstr}(X)$ iff all $\delta \in \mathbb{R}^{\geq 0}$ and all continuously differentiable functions $\varphi : [0, \delta] \to \mathbb{R}^X$ with $\varphi(0) \in [\![\mathsf{Init}]\!]$ also satisfy $\varphi(\delta) \in [\![\mathsf{Safe}]\!]$ provided that $\dot{\varphi}(t) = f(\varphi(t))$ and $f(t) \in [\![\mathsf{Inv}]\!]$ for all $t \in [0, \delta]$. We also say that the continuous system $(\mathsf{Init}, \dot{x} = f(x), \mathsf{Inv})$ *respects* Safe if $(\mathsf{Init}, \dot{x} = f(x), \mathsf{Inv})$ is safe with respect to property Safe.

Logic-based verification techniques provide easily checkable witnesses to verify that a continuous system $(\mathsf{Init}, \dot{x} = f(x), \mathsf{Inv})$ respects Safe. The immediate significance for model checking is that they induce abstractions that can be used to terminate a reachability computation.

Lemma 6 (Logical abstraction) *Let* $(\mathsf{Init}, \dot{x} = f(x), \mathsf{Inv})$ *be a continuous system of a mode* $\ell \in \mathsf{Loc}$ *of a hybrid automaton. If* $(\mathsf{Init}, \dot{x} = f(x), \mathsf{Inv})$ *respects* Safe, *then*

$$\mathsf{post}_\ell\big([\![\mathsf{Init}]\!]\big) \subseteq [\![\mathsf{Safe}]\!].$$

If the continuous system $(\mathsf{Init}, \dot{x} = f(x), \mathsf{Inv})$ of a mode $\ell \in \mathsf{Loc}$ of a hybrid automaton respects the desired safety property Safe, (continuous) reachability computation can be terminated for all states in any subset $P \subseteq [\![\mathsf{Init}]\!]$, because, by monotonicity, Lemma 6 then implies

$$\mathsf{post}_\ell(P) \subseteq [\![\mathsf{Safe}]\!].$$

In particular, notice that it is useful for fast reachability computation if we can identify big sets Init that make $(\mathsf{Init}, \dot{x} = f(x), \mathsf{Inv})$ respect Safe. These sets Init are often much bigger than the original initial sets from Definition 6.

The logic-based verification techniques mentioned above have in common that they provide easily checkable witnesses for the verification. They further enjoy the benefit that they can be used for highly nonlinear dynamics. The primary challenge in all cases is the need to first find the witnesses or their shape, which corresponds to the challenge of finding the right directions for support functions.

An interesting special case of the continuous safety problem from Definition 12 is the case where Init and Safe are the same formula F. If the continuous system $(F, \dot{x} = f(x), \mathsf{Inv})$ is safe with respect to F, then F is called a (safety) *invariant*. In that case, Lemma 6 implies

$$\mathsf{post}_\ell\big([\![F]\!]\big) \subseteq [\![F]\!].$$

That is, the continuous system will never be able to leave F. Thus, without reachability computation, one can conclude that reachable sets that are within $[\![F]\!]$ will stay there forever.

Observe that, despite the similar name, there is a crucial difference between an invariant condition Inv of a continuous system (or a mode in a hybrid system) and

a safety invariant F. The difference is that we need to verify whether F is a safety invariant, while we just assume that the system obeys the invariant condition Inv. That is why Inv is also called an evolution domain restriction, because it restricts the admissible evolution domain of the continuous system. So, Inv is part of the system model, yet F is part of a safety property that we verify for the system model.

One of many possible approaches to logic-based verification is the one that focuses on showing that a formula F is a global invariant of a hybrid automaton by showing that it is an invariant for each discrete transition and an invariant for each continuous transition of the automaton. The best case is if F is the safety property and turns out to be a global invariant of the system in this manner. This is generally somewhat overly simplistic, because the verification does not necessarily have to work with the same invariant F in all places so that multiple invariants need to be used instead. Nevertheless, having this simple example of a single global invariant in mind is a useful guiding principle for logic-based verification approaches.

Related arguments have also been used for invariant generation techniques for abstract interpretation [169]. Based on decidability results for o-minimal hybrid automata [109], this includes invariant generation techniques for linear systems based on Gröbner basis computations [169] rather than based on quantifier elimination [109]. The case of (hyper-rectangle) box invariants has been discussed in more detail elsewhere [170].

30.6.1 Polynomial Barrier Certificates

The basic idea behind barrier certificates is to find a barrier separating good and bad states that we can easily show to be impenetrable by the continuous system dynamics. Barrier certificates were proposed for safety verification in [154].

Theorem 4 (Weak barrier certificate [154]) *Let* $(\mathsf{Init}, \dot{x} = f(x), \mathsf{Inv})$ *be a continuous system with safety constraint* Safe. *If* B *is a* (weak) barrier certificate *for a continuous safety problem, i.e., a polynomial satisfying*

$$\begin{aligned} B(x) &\leq 0 \quad \textit{for all initial states } x \in [\![\mathsf{Init}]\!], \\ B(x) &> 0 \quad \textit{for all unsafe states } x \notin [\![\mathsf{Safe}]\!], \quad \textit{and} \\ \frac{\partial B}{\partial x}(x) f(x) &\leq 0 \quad \textit{for all states } x \in [\![\mathsf{Inv}]\!], \end{aligned}$$

then the continuous system $(\mathsf{Init}, \dot{x} = f(x), \mathsf{Inv})$ *respects* Safe.

Barrier certificates themselves can be defined for more general non-polynomial cases, but the conditions are generally not computable when $f_i(x)$ and B are not polynomials or Inv, Init, and Safe are not polynomial constraints. The purpose of a barrier certificate is to separate safe from unsafe states in such a way that initial

states are safe, and the differential equations can easily be seen to never cross the barrier between safe and unsafe states.

The importance of barrier certificates comes from the fact that they reduce a reachability question (can we ever reach an unsafe state) to a simple check on the directional derivative $\frac{\partial B}{\partial x}(x)\,f(x)$ along ODE of the Barrier certificate.

It had originally been proposed [153] that barrier certificates only need to be checked on the boundary of the barrier and that it would be sufficient to check the third condition in Theorem 4 for all $x \in [\![\mathsf{Inv}]\!]$ with $B(x) = 0$:

$$\frac{\partial B}{\partial x}(x)\,f(x) \leq 0 \quad \text{for all states } x \in [\![\mathsf{Inv}]\!] \text{ with } B(x) = 0. \tag{16}$$

This condition is generally not strong enough and can lead to soundness issues, as the following example shows.

Example 2 When using condition (16), it looks as if the differential equation $\dot{x} = 1$ always stays in the region $\mathsf{Safe} \equiv x^2 \leq 0$ because condition (16) succeeds as follows:

$$\frac{\partial x^2}{\partial x} 1 = 2x \leq 0 \quad \text{for all states } x \text{ with } x^2 = 0$$

This, however, is counterfactual, because the system $\dot{x} = 1$ will, of course, leave region $x^2 \leq 0$. Thus, the condition (16) is unsound. The same issue occurs for a suggestion on how to extend this approach to Boolean combinations of inequalities [84]. A discussion of the assumptions under which the conditions can be restricted to such subsets without losing soundness can be found in the literature [136, 142, 144, 154].

Checking on the boundary is sound, however, if the condition (16) is modified to a strict inequality, instead of a weak inequality:

Theorem 5 (Strict barrier certificate [154]) *Let* $(\mathsf{Init}, \dot{x} = f(x), \mathsf{Inv})$ *be a continuous system with safety constraint* Safe. *If* B *is a* (strict) barrier certificate *for a continuous safety problem, i.e., a polynomial satisfying*

$$\begin{aligned} B(x) &\leq 0 \quad \textit{for all initial states } x \in [\![\mathsf{Init}]\!], \\ B(x) &> 0 \quad \textit{for all unsafe states } x \notin [\![\mathsf{Safe}]\!], \quad \textit{and} \\ \frac{\partial B}{\partial x}(x)\,f(x) &< 0 \quad \textit{for all states } x \in [\![\mathsf{Inv}]\!] \textit{ with } B(x) = 0, \end{aligned}$$

then the continuous system $(\mathsf{Init}, \dot{x} = f(x), \mathsf{Inv})$ *respects* Safe.

Search procedures for barrier certificates include approaches that choose a degree-bound for the barrier certificate $B(x)$ and then turn the conditions from Theorem 4 into a convex optimization problem, which can be solved efficiently [154].

A similar approach has been proposed for Theorem 5, but the optimization problem is then non-convex [154], so optimizers can get stuck in local optima.

Barrier certificates can be extended to systems with disturbances and to switching diffusion systems [154]. We refer to the literature for a discussion of these generalizations and examples [154].

30.6.2 Equational Certificates

Equational certificates [159, 161] serve a purpose that has quite some similarity to barrier certificates. They were introduced [161] at the same time as barrier certificates [154], and later rephrased and generalized [159] similarly to a matrix reformulation of that idea [123]. Equational certificates have been investigated earlier by Darboux in 1878 [56] for continuous systems not in the context of hybrid systems. Like barrier certificates, the conditions of equational certificates make a reachability analysis superfluous, because they give a simple certificate showing a property of the system. One major difference of equational certificates compared to barrier certificates is that an equational certificate consists of a single polynomial *equation* $p(x) = 0$, while a barrier certificate consists of a single polynomial *inequality* $B(x) \leq 0$. The other major difference is the condition itself. It is an equational criterion, not using inequalities. Another minor difference is that an equational certificate $p(x) = 0$ shows invariance of the property $p(x) = 0$ instead of separating initial states from bad states. That is a minor difference, though, because Safe is an invariance property that can be read off from a barrier certificate that separates Init from ¬Safe.

Theorem 6 (Equational certificates [161]) *Let $p(x)$ be a polynomial and let $(p(x) = 0, \dot{x} = f(x), \mathsf{Inv})$ be a continuous system. If there is a polynomial $g(x)$ such that*

$$\frac{\partial p}{\partial x}(x) f(x) = g(x) p(x)$$

for all $x \in [\![\mathsf{Inv}]\!]$, then $(p(x) = 0, \dot{x} = f(x), \mathsf{Inv})$ respects $p(x) = 0$. In particular, $p(x) = 0$ is an invariant of $(p(x) = 0, \dot{x} = f(x), \mathsf{Inv})$.

The equational template approach for equational certificates [161] works as follows. The user chooses a template for the polynomial equation $p(x) = 0$ and the system then uses linear equation solving and/or Gröbner basis computations [38] to check whether the equational certificate condition from Theorem 6 holds. In general, the approach may use the decision procedures of quantifier elimination in real closed fields [49] to handle the nonlinear real arithmetic.

Common special cases of equational certificates include those where only numbers or only 0 is chosen for the polynomial $g(x)$. It had originally been proposed

informally [161] that it should also be sufficient in Theorem 6 to check

$$\frac{\partial p}{\partial x}(x)\, f(x) = 0 \quad \text{for all } x \in [\![\mathsf{Inv}]\!] \text{ with } p(x) = 0. \tag{17}$$

This variation is generally not strong enough and can lead to soundness issues.

Example 3 When using condition (17), it may seem as if $x^2 = 0$ were an invariant of the differential equation $\dot{x} = 1$, because condition (17) succeeds as follows:

$$\frac{\partial x^2}{\partial x} 1 = 2x = 0 \quad \text{for all } x \text{ with } x^2 = 0$$

This, however, is counterfactual, because the system $\dot{x} = 1$ will, of course, falsify the safety condition $x^2 = 0$ right away. Thus, the condition (17) is an unsound variation of Theorem 6. We refer to the literature [136, 142] for a discussion of the conditions under which stronger assumptions can be assumed without losing soundness.

There are additional conditions on the system dynamics and p, however, under which the restriction (17) remains correct [142]. That line of research also identifies under which conditions equational templates and equational differential invariants are complete for verifying equational safety properties [142].

30.6.3 Differential Invariants and Logical Certificates

Differential invariants are a generalized form of logic-based witness techniques for hybrid systems and generalize equational certificates [161] and barrier certificates [153, 154]. Like equational certificates [161], a differential invariant can be an equation $p(x) = 0$. Like barrier certificates [153, 154], differential invariants can be inequalities like $p(x) \leq 0$. Differential invariants can be general logical formulas with propositional combinations of mixed equations, strict inequalities, and weak inequalities, and can be extended to contain quantifiers for distributed hybrid systems [138]. Differential invariants have been introduced in 2008 [136] and later refined to an automatic verification procedure that searches for differential invariants [148]. Further results about the theory of differential invariants can be found in the literature [142, 144].

Given a continuous system $(\mathsf{Init}, \dot{x} = f(x), \mathsf{Inv})$, we want to check whether it respects Safe. As a short notation, we say that the formula $\mathsf{Init} \rightarrow [\dot{x} = f(x)\&\mathsf{Inv}]\mathsf{Safe}$ *is valid* if the continuous system $(\mathsf{Init}, \dot{x} = f(x), \mathsf{Inv})$ respects the safety condition Safe. That is, if that continuous system always stays in the region Safe when it follows differential equation $\dot{x} = f(x)$ restricted to the evolution domain region Inv and when started in any initial state satisfying Init. Even though more complex representations can be used, we assume Init, Safe, and Inv to be (semi-algebraic) polynomial constraints. A simple form corresponds to the case where Init and Safe are the same

formula F. If $F \rightarrow [\dot{x} = f(x)\&\mathsf{Inv}]F$ is valid, then F is called a *continuous invariant* of the dynamics $\dot{x} = f(x)\&\mathsf{Inv}$. That is, if the continuous system starts in F, then it will always stay in F.

In fact, the notation $\mathsf{Init} \rightarrow [\dot{x} = f(x)\&\mathsf{Inv}]\mathsf{Safe}$ can be understood as a logical formula. The logical formula $[\dot{x} = f(x)\&\mathsf{Inv}]\mathsf{Safe}$ uses the modal operator $[\dot{x} = f(x)\&\mathsf{Inv}]$ to say that formula Safe holds in all states that are reachable along the differential equation $\dot{x} = f(x)$ within evolution domain Inv. The implication $\mathsf{Init} \rightarrow$ in $\mathsf{Init} \rightarrow [\dot{x} = f(x)\&\mathsf{Inv}]\mathsf{Safe}$ restricts this to only the set of initial states that satisfy Init. The same principle extends to a logic for hybrid systems [135–137, 141, 143] and to a logic for distributed hybrid systems [140]; see [143] for an overview. Both of these logics are relatively complete (similarly to relative completeness of Hoare calculus). That is, they can prove every valid formula about hybrid systems or (distributed) hybrid systems from elementary properties of differential equations. These results also give a precise construction lifting all verification techniques for continuous systems to hybrid systems [141].

Differential invariants can be equational formulas like equational certificates, they can include inequalities like barrier certificates, but they also include mixed cases, Boolean combinations, and cases with more complicated logical formulas.

Definition 13 (Continuous invariant) Let $(\mathsf{Init}, \dot{x} = f(x), \mathsf{Inv})$ be a continuous system with safety constraint Safe. Constraint F is a *continuous invariant* of $\mathsf{Init} \rightarrow [\dot{x} = f(x)\&\mathsf{Inv}]\mathsf{Safe}$ iff the following formulas are valid (true in all states):

1. $\mathsf{Init} \wedge \mathsf{Inv} \rightarrow F$ (induction start), and
2. $F \rightarrow [\dot{x} = f(x)\&\mathsf{Inv}]F$ (induction step).

A continuous invariant F is *sufficiently strong* for $\mathsf{Init} \rightarrow [\dot{x} = f(x)\&\mathsf{Inv}]\mathsf{Safe}$ if, in addition, $F \rightarrow \mathsf{Safe}$ is valid, because $\mathsf{Init} \rightarrow [\dot{x} = f(x)\&\mathsf{Inv}]\mathsf{Safe}$ is then valid.

It is easy to see that the existence of a sufficiently strong continuous invariant for $\mathsf{Init} \rightarrow [\dot{x} = f(x)\&\mathsf{Inv}]\mathsf{Safe}$ implies that the property $\mathsf{Init} \rightarrow [\dot{x} = f(x)\&\mathsf{Inv}]\mathsf{Safe}$ is valid.

Continuous invariants are useful notions, but they are not computational per se, because we still need to find a way to check the induction step. The induction start is reasonable, because it is just a constraint, which is a logical formula of first-order real arithmetic and thus decidable by quantifier elimination in real closed fields [49, 50, 166]. But we need to find a checkable representation of the induction step. A checkable condition is made formally precise using the notion of differential invariants.

Definition 14 (Differential invariant) Let $(\mathsf{Init}, \dot{x} = f(x), \mathsf{Inv})$ be a continuous system with safety constraint Safe. A polynomial constraint F is a *differential invariant* of $\mathsf{Init} \rightarrow [\dot{x} = f(x)\&\mathsf{Inv}]\mathsf{Safe}$ iff the following formulas are valid:

1. $\mathsf{Init} \wedge \mathsf{Inv} \rightarrow F$ (induction start), and
2. $\mathsf{Inv} \rightarrow \nabla_{\dot{x}=f(x)}F$ (induction step),

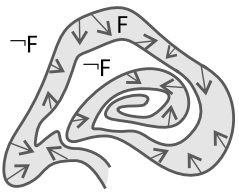


Fig. 13 Differential invariant F

where $\nabla_{\dot{x}=f(x)}F$ is the conjunction of all directional derivatives of atomic formulas in F in the direction of the vector field of $\dot{x} = f(x)$ (the partial derivative of b by x_i is $\frac{\partial b}{\partial x_i}$):

$$\nabla_{\dot{x}=f(x)}F \equiv \bigwedge_{(b\sim c)\in F}\left(\sum_{i=1}^{n}\frac{\partial b}{\partial x_i}f_i(x)\right) \sim \left(\sum_{i=1}^{n}\frac{\partial c}{\partial x_i}f_i(x)\right)$$

where $\sim \in \{=, \geq, >, \leq, <\}$.

A differential invariant F is *sufficiently strong* for $\mathsf{Init} \to [\dot{x} = f(x)\&\mathsf{Inv}]\mathsf{Safe}$ if, in addition, $F \to \mathsf{Safe}$ is valid (because $\mathsf{Init} \to [\dot{x} = f(x)\&\mathsf{Inv}]\mathsf{Safe}$ is then valid by Corollary 2 below).

The respective partial derivatives of terms are well defined in the Euclidean space spanned by the variables and can be computed symbolically [136, 137]. Differential invariants capture the condition showing that the formula F is only becoming more true when following the dynamics, not less true, see Fig. 13.

The central property of differential invariants for verification purposes is that they replace infeasible or impossible reachability analysis with feasible symbolic computation.

Theorem 7 (Principle of differential induction [136]) *All differential invariants are continuous invariants.*

Corollary 1 *If F is a differential invariant for* $\mathsf{Init} \to [\dot{x} = f(x)\&\mathsf{Inv}]\mathsf{Safe}$, *then* $\mathsf{Init} \to [\dot{x} = f(x)\&\mathsf{Inv}]F$ *is valid.*

Corollary 2 *If F is a differential invariant for* $\mathsf{Init} \to [\dot{x} = f(x)\&\mathsf{Inv}]\mathsf{Safe}$ *that is sufficiently strong, then F is a continuous invariant that is sufficiently strong for* $\mathsf{Init} \to [\dot{x} = f(x)\&\mathsf{Inv}]\mathsf{Safe}$. *In particular,* $\mathsf{Init} \to [\dot{x} = f(x)\&\mathsf{Inv}]\mathsf{Safe}$ *is valid.*

Example 4 Consider the dynamics $\dot{x} = x^4, \dot{y} = -2$. We are interested in seeing whether $2x \geq 5y$ is an invariant of this dynamics. With differential invariants it is easy to show that this is an invariant for the dynamics without using any state-based reachability verification. We just compute symbolically:

$$\nabla_{\dot{x}=x^4,\dot{y}=-2}(2x \geq 5y) \equiv \frac{\partial 2x}{\partial x}x^4 + \frac{\partial 2x}{\partial y}(-2) \geq \frac{\partial 5y}{\partial x}x^4 + \frac{\partial 5y}{\partial y}(-2) \equiv 2x^4 \geq -10.$$

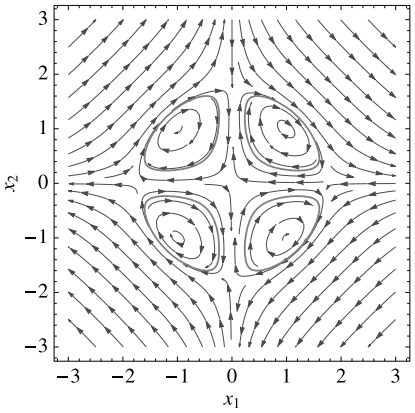


Fig. 14 Example of a differential invariant indicated by the thick boundary

Since the latter formula is easily found to be valid, $2x \geq 5y$ is proven to be a differential invariant and thus stays true whenever it holds for the initial state of the dynamics.

Consider the case where $\dot{x} = x^4$, $\dot{y} = -2$ is the dynamics of one location of a hybrid automaton. Then we know that $2x \geq 5y$ is true after staying in this location arbitrarily long, if only we know that $2x \geq 5y$ is also true initially when entering the location. This is a prototypical scenario where local verification results also need to be combined together in order to verify the whole hybrid automaton.

Example 5 Consider the dynamics

$$\dot{x_1} = 2x_1^4x_2 + 4x_1^2x_2^3 - 6x_1^2x_2, \qquad \dot{x_2} = -4x_1^3x_2^2 - 2x_1x_2^4 + 6x_1x_2^2.$$

Using differential invariants it is easy to show that $x_1^4x_2^2 + x_1^2x_2^4 - 3x_1^2x_2^2 + 1 \leq c$ is an invariant of this dynamics, as illustrated in Fig. 14. The justification again follows by simple symbolic computation as in Example 4:

$$\begin{aligned}
&\nabla_{\dot{x_1}=2x_1^4x_2+4x_1^2x_2^3-6x_1^2x_2,\dot{x_2}=-4x_1^3x_2^2-2x_1x_2^4+6x_1x_2^2}\left(x_1^4x_2^2 + x_1^2x_2^4 - 3x_1^2x_2^2 + 1 \leq c\right)\\
&\quad\equiv \frac{\partial\left(x_1^4x_2^2 + x_1^2x_2^4 - 3x_1^2x_2^2 + 1\right)}{\partial x_1}\left(2x_1^4x_2 + 4x_1^2x_2^3 - 6x_1^2x_2\right)\\
&\qquad + \frac{\partial\left(x_1^4x_2^2 + x_1^2x_2^4 - 3x_1^2x_2^2 + 1\right)}{\partial x_2}\left(-4x_1^3x_2^2 - 2x_1x_2^4 + 6x_1x_2^2\right) \leq 0
\end{aligned}$$

which simplifies to true.

Differential invariants work somewhat like loop invariants but for differential equations instead of loops. When checking a loop invariant F, we can assume it holds before the loop in the induction step. It thus looks as if we should be able to assume F when proving the induction step Case 2 of Definition 14 and prove

$$\mathsf{Inv} \wedge F \rightarrow \nabla_{\dot{x}=f(x)} F \tag{18}$$

instead. Or, better, yet, only check the condition on the boundary of the domain like for barrier certificates. Neither of those would be sound, however, according to the following counterexamples from [136, 148]:

Example 6 When using condition (18), it looks as if $x^2 \leq 0$ were an invariant of the differential equation $\dot{x} = 1$, because condition (18) succeeds as follows:

$$\left(x^2 \leq 0 \rightarrow \nabla_{\dot{x}=1} x^2 \leq 0\right) \equiv \left(x^2 \leq 0 \rightarrow \frac{\partial x^2}{\partial x} 1 \leq 0\right) \equiv \left(x^2 \leq 0 \rightarrow 2x \leq 0\right)$$

This, however, is counterfactual, because the system $\dot{x} = 1$ will, of course, leave region $x^2 \leq 0$. Thus, the condition (18) is unsound. The same example shows that checking on the boundary of F is unsound in general. We refer to the original work [136] for a discussion of the conditions under which stronger assumptions can be made without losing soundness.

A further elaboration of these phenomena as well as an identification of the conditions under which such extra assumptions would be sound can be found in the literature [142, 144].

It turns out that some properties cannot be verified using differential invariants alone but that additional verification techniques are needed [144]. Differential saturation (repeated application of differential cuts [136, 144]) has been introduced together with differential invariants in 2008 [136] as a sound alternative that can be used to add conditions iteratively without compromising soundness.

Theorem 8 (Differential saturation [136, 144]) *Assume that F is a continuous invariant (e.g., a differential invariant) of* $\mathsf{Init} \rightarrow [\dot{x} = f(x)\&\mathsf{Inv}]\mathsf{Safe}$, *then*

$$\mathsf{Init} \rightarrow [\dot{x} = f(x)\&\mathsf{Inv}]\mathsf{Safe} \quad \textit{iff} \quad \mathsf{Init} \rightarrow [\dot{x} = f(x)\&\mathsf{Inv} \wedge F]\mathsf{Safe}.$$

An evolution domain constraint Inv (also confusingly referred to as the invariant of a location) is an entirely different entity than an invariant property F of a system. An automaton model *assumes or prescribes* that the system dynamics can only be followed along traces that do not leave Inv, because the system will stop all executions that leave Inv. In contrast, a differential invariant *proves* that the system will never leave F whether it wants to or not. Nevertheless, Theorem 8 gives a sound way of translating a proved differential invariant into a prescriptive evolution domain constraint. Theorem 8 can be used to strengthen the evolution domain constraints to subregions, which then become available for subsequent verification in a sound way. The differential cut principle underlying Theorem 8 is particularly powerful when used repeatedly until saturation [136, 148]. That is, verification with differential invariants often proceeds in stages, where a number of formulas F are verified to be invariants and then used to constrain the evolution of the system using the right-hand side of Theorem 8. This process repeats until all unsafe states have

been verified to be removable from the state space and so verification becomes trivial. Repeating this process in a fixed-point loop has been shown to work successfully in practice [148].

Differential invariants are computationally attractive concepts, because their induction start and induction step are just polynomial constraints, which are formulas of first-order real arithmetic, and are thus decidable by quantifier elimination in real closed fields [49, 50, 166]. Also the check whether a differential invariant F is sufficiently strong to imply a polynomial safety constraint Safe is decidable. The steps needed to compute the induction step of a differential invariant are simple algebraic computations that can be automated easily.

Differential invariants are always sound. That is, every property that can be verified using a differential invariant is correct. The converse question is that of completeness, whether all relevant properties can be verified. It turns out that differential invariants alone are not complete.

Example 7 $x > 0 \rightarrow [\dot{x} = -x \& true]x > 0$ is valid, but $x > 0$ is not a differential invariant of $\dot{x} = -x$, not a barrier certificate, and does not qualify as an equational template either.

More generally, it can be shown that there are properties like Example 7 that are true but cannot be verified [144], except when using an additional verification technique known as differential auxiliaries (alias differential ghosts) that adds additional variables and additional dynamics for verification purposes [144]. Thus, differential auxiliaries are a fundamental extension that is required for verification.

Search procedures for differential invariants include degree-bounded enumeration and fixed-point loops [148]. For completeness guarantees and numerous provability relationships on classes of differential invariants, see [141, 144]. The case of equational differential invariants is elaborated in [142], in which case differential invariants are a necessary and sufficient criterion for invariant functions according to a corresponding result by Lie.

Theorem 9 (Invariant function characterization) *A (polynomial) function p is an invariant function of $\dot{x} = f(x)$, i.e., the value of p along all solutions is constant, iff $p = 0$ is a differential invariant of $\dot{x} = f(x)$.*

A corresponding necessary and sufficient characterization of all algebraic invariant equations of algebraic differential equations is possible with a higher-order generalization of equational differential invariants called *differential radical invariants* [73].

For hybrid systems, differential invariants are used by allowing separate invariants for the respective locations of the hybrid automaton. Consider the hybrid automaton in Fig. 15 and, for the moment, suppose that there are no discrete jumps, i.e., the reset relations are the identity relation. Then, we need to show that starting in F_1 for dynamics $\dot{x} = \theta_1$ will always stay in the region F_2. In addition, we need to show that, when starting in F_2 the dynamics $\dot{x} = \theta_2$ will always stay in

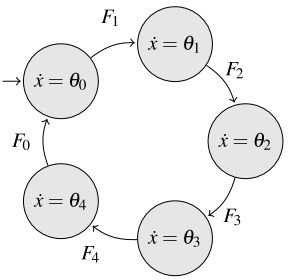


Fig. 15 Example of a verification loop for a hybrid automaton

the region F_3, and so on. That is, in general we need to show that, when starting in F_i, the dynamics $\dot{x} = \theta_i$ will always stay in the region $F_{(i+1)\%5}$. In the presence of non-trivial discrete jump relations, we also need to show that these jump relations preserve the respective invariant. That means, we need to show that the jump relation (including its guard) will always transform every state within the invariant region F_i of its source into the invariant region $F_{(i+1)\%5}$ of its target. Finally, we only know that the reachable states of the hybrid automaton are contained in the respective invariant regions F_i if the automaton also starts in the required invariant region F_0 of the initial location. That is, we need to check that the initial state is contained in F_0.

To make this principle concrete, consider a flyable roundabout maneuver for air traffic control [149], which is a variation of roundabouts that have been proposed a decade before [175]. Flyable roundabouts follow a hybrid automaton similar to Fig. 15, but with locations that correspond to the various phases of the roundabout as depicted schematically in Fig. 16. The aircraft are initially in free flight (free), then, when a conflict arises, agree on a compatible roundabout collision avoidance maneuver (agree), approach the roundabout with an entry procedure (entry), follow the roundabout (circ), and then leave the roundabout (exit), until they are far enough away to enter free flight again. Such roundabout collision avoidance maneuvers for aircraft can be verified using differential invariants, see elsewhere [149] for details.

For an investigation of the theory of differential invariants, we refer to [73, 136, 142–144]. That line of research studies the theoretical and provability properties of differential invariants. It identifies a dozen relations either equating or separat-

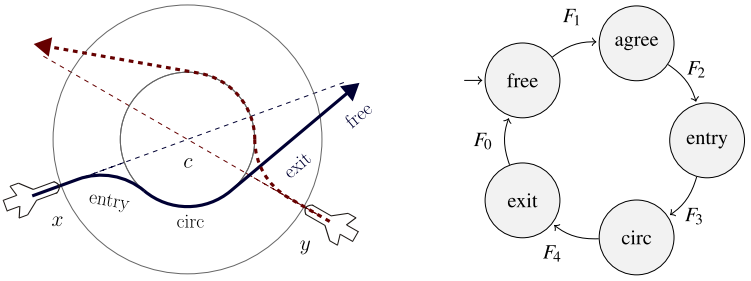
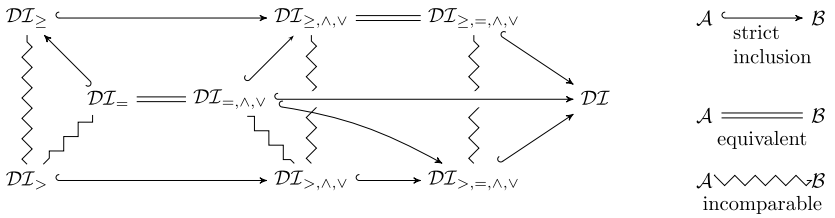


Fig. 16 Phases of flyable roundabout maneuver and protocol cycle



$\mathcal{DI}_{\Omega}$: properties verifiable using differential invariants built with operators from Ω

Fig. 17 Differential invariance chart: identifies how classes of differential invariants relate to each other, where the operators in the differential invariants are restricted as indicated in subscript Ω

ing the verification power of various classes of differential invariants (Fig. 17 indicates strict inclusion, equivalence, and incomparability of verification power, respectively). These relations further imply that the inclusion of Boolean operators that differential invariants support makes it possible to verify more systems compared to the single polynomial inequalities of barrier certificates or the single polynomial equations of equational templates [144]. The subclass of systems that have equational systems as invariants, however, already have a single equational invariant [144]. Differential cuts, differential saturation, and differential auxiliaries have been identified as fundamental extensions [136, 144]. The surprisingly close relationship of differential invariants to classical discrete invariants has been explored in the literature [141]. The relationship of differential invariants to Lie's seminal work, a differential operator view, and partial differential equations has been investigated along with a technique called the inverse characteristic method for generating differential invariants [142]. The generalization of differential radical invariants can be generated efficiently using symbolic linear algebra [73].

For a generalization of differential invariants to systems with disturbances and differential-algebraic equations, we refer to the literature [136, 137]. Differential invariants can be generalized to the case of quantified first-order formulas and to distributed hybrid systems [138]. The approach extends to a relatively complete logic for hybrid systems [135–137, 141, 143] and to a relatively complete logic for distributed hybrid systems [140]. Generalizations to reachability and progress conditions can be found elsewhere [136]. Generalizations to stochastic hybrid systems with stochastic differential equations have been proposed [139].

30.7 Verification Tools

Despite the undecidability of the general case, the safety verification problem has been attacked algorithmically: many of the classical tools (among others D/DT [19], CheckMate [45], HyTech [89]) and many of the more recent tools (PHAVer [67]) use a symbolic analysis of the hybrid automaton with a forward and/or backward approach: starting from the initial (resp. unsafe) states, iterate the Post operator (resp.

Pre) until a fixed point is reached and then check emptiness of the intersection with the unsafe (resp. initial) states. By Theorem 2, these procedures are not guaranteed to terminate in general. As discussed in Sect. 30.4, a major issue is scalability, as the computational cost increases sharply with the number of continuous variables. Performance is achieved by overapproximating the Post operator, and overapproximation can also be used to force termination of the fixed-point procedure. The challenge is to find methods that scale and are still accurate enough to show safety.

We discuss a selection of hybrid systems verification tools representing different classes of approaches that we survey here. A complete overview of all tools is beyond the scope of this chapter. We focus on a subset of the verification tools for which a dedicated tool paper and at least some documentation is available. A more complete collection of tools can be found on the Web.[8]

HSolver: Interval Constraint Propagation

HSolver[9] [158] is an open-source software package for the formal verification of safety properties of continuous-time hybrid systems. It allows hybrid systems with nonlinear ordinary differential equations and nonlinear jumps assuming a global compact domain restriction on all variables. Even though HSolver is based on fast machine-precision floating point arithmetic, it uses sound rounding, and hence the correctness of its results cannot be hampered by round-off errors. HSolver not only verifies (unbounded horizon) reachability properties of hybrid systems, but—in addition—it also computes abstractions of the input system. So, even for input systems that are unsafe, or for which exhaustive formal verification is too difficult, it will compute abstractions that can be used by other tools. For example, the abstractions could be used for guiding search for error trajectories of unsafe systems.

HSolver is not optimized for special classes of hybrid systems (e.g., systems such as linear hybrid automata that have very simple continuous dynamics). Moreover it does not yet provide mature support for finding counterexamples for unsafe input systems. The method used by HSolver is abstraction refinement based on interval constraint propagation [158], which incrementally refines an abstraction of the input system. Special care is taken to reflect as much information as possible in the abstraction without increasing its size.

HyTech: The HYbrid TECHnology Tool

HyTech[10] [89] was the first tool for reachability analysis of PCDA (Linear Hybrid Automata). The system is specified as a product of automata that synchronize

[8] http://wiki.grasp.upenn.edu/.

[9] http://hsolver.sourceforge.net/.

[10] http://embedded.eecs.berkeley.edu/research/hytech/.

on transitions that share the same label. The tool has a simple command language similar to a basic imperative language, allowing the user to program his own exploration algorithms. The basic data type represents a union of polyhedra associated with each location of the product automaton. Operations such as Boolean operations, existential quantification, emptiness test, and reachability computation (using the Post operation) are provided. Error traces (counterexamples) can be produced in combination with reachability analysis.

HyTech uses polyhedra with the double description method, which combines constraint and generator representations. The post operators are those described in Sect. 30.4.2 and implemented with exact arithmetic. HyTech can be used for parametric analysis by viewing parameters as variables with first derivative equal to zero. For instance, existential quantification on the reachable states can be used to extract a constraint on the parameters such that a given region is reachable. HyTech has been used to model check an audio control protocol [99] and a steam boiler [98]. A main limitation of HyTech lies in its use of standard integer data types, which quickly leads to integer overflow.

KeYmaera: Logic and Differential Invariants for Compositional Verification

KeYmaera[11] [135–137, 141, 150] is a hybrid verification tool for hybrid systems that combines deductive, real algebraic, and computer algebraic prover technologies. It is an automated and interactive theorem prover for a natural specification and verification logic for hybrid systems. With this, the verification principle behind KeYmaera is fundamentally different and complementary to tools like HyTech [89], PHAVer [67], and SpaceEx [68]. KeYmaera supports differential dynamic logic (d$\mathcal{L}$) [135, 137, 141], which is a real-valued first-order dynamic logic for hybrid programs [135, 137, 141], a program notation for hybrid systems. KeYmaera also supports hybrid systems with nonlinear discrete jumps, nonlinear differential equations, differential-algebraic equations, differential inequalities, and systems with nondeterministic discrete or continuous input.

For automation, KeYmaera implements a number of automated proof strategies that decompose the hybrid system symbolically and prove properties of the full system by proving properties of its parts [137]. This compositional verification principle helps scale up verification, because KeYmaera verifies large systems by verifying properties of subsystems (also see assume-guarantee reasoning, Chap. 12). KeYmaera implements fixed-point procedures [148] that compute differential invariants and invariants in fixed-point loops, somewhat like the way classical model checkers compute reachable sets in fixed-point loops. KeYmaera is typically more suitable for verifying parametric hybrid systems than systems with a single numerical state, where simulation is more appropriate. KeYmaera has been used successfully for verifying case studies in train control [151], car control [114, 115], air

[11] http://symbolaris.com/info/KeYmaera.html.

traffic management [149], mobile robotics [130] and surgical robotics [104]. The KeYmaera approach is described in a book about Logical Analysis of Hybrid Systems [137].

A comprehensive introduction is provided in a textbook on the logical foundations of cyber-physical systems [146]. This textbook also explains how the successor tool KeYmaera[12] [72] can achieve the same from a minimal soundness-critical core [145].

PHAVer: Polyhedral Hybrid Automaton Verifyer

PHAVer[13] [67] follows the same basic principles as HyTech. PHAVer is a formal verification tool for computing reachability and simulation relations of PCDA (Linear Hybrid Automata) from Sect. 30.4.2.

PHAVer uses standard operations on polyhedra for reachability computations over an infinite time horizon (similar to those used in HyTech), and the algorithm for computing simulation relations is a straightforward extension of these. Using unbounded integer arithmetic, the computations are exact and formally sound. In addition to PCDA reachability, PHAVer can overapproximate piecewise affine dynamics on the fly, computing an overapproximation of the reachable states that is invariant (all trajectories that start within the set stay within the set). While PCDA are undecidable, PHAVer provides formally sound and precise overapproximation and widening operators that can force termination at the cost of reduced precision. These operators also simplify the computed continuous sets and dynamics of the system, and may result in a considerable speed-up without much loss in precision. The checking of abstraction and equivalence with simulation relations can be applied compositionally, and a sound non-circular assume-guarantee rule is implemented [66]. However, since the required exact computations on polyhedra do not scale well, this approach is limited to very small systems.

With its exact computations and controllable overapproximations, PHAVer is suited to verifying formally stringent properties on small systems with simple dynamics, such as communication protocols with drifting clocks or buffer networks. PHAVer's disadvantage is that the employed polyhedra computations are generally exponential in the number of variables, so that scalability is limited. PHAVer has been used to verify oscillation properties of a voltage-controlled oscillator circuit with three state variables [70], and various academic benchmarks with simple dynamics and up to 14 continuous variables. Since 2011, PHAVer is part of the SpaceEx tool platform [68].

[12] http://keymaeraX.org/.

[13] http://www-verimag.imag.fr/~frehse/phaver_web/index.html.

SpaceEx: State Space Explorer

SpaceEx[14] [68] is a tool platform for verifying hybrid systems. It can handle hybrid automata whose continuous and jump dynamics are piecewise affine with nondeterministic inputs, i.e., PWA from Sect. 30.4.3. Nondeterministic inputs are particularly useful for modeling the approximation error when nonlinear systems are brought into piecewise affine form. SpaceEx comes with a compositional modeling language. It allows one to specify complex systems in a modular fashion as a network of interacting hybrid automata with templates and nesting. In the SpaceEx model editor, components are connected in block diagrams known from control theory, and the evolution of continuous variables is specified by hybrid automata with differential algebraic equations and inequalities.

Several different algorithms are implemented on the SpaceEx platform, including an exact algorithm for PCDA and a simulator that can handle nonlinear dynamics. The main verification algorithms, called LGG [68] and STC [69], combine explicit set representations (polyhedra), implicit set representations (support functions), and linear programming to achieve high scalability while maintaining high accuracy. The reachable states are overapproximated in the form of template polyhedra, which are polyhedra whose facets are oriented according to a user-provided set of template directions. The algorithms use adaptive time steps to ensure that the approximation error in each template direction remains below a given value. Empirical measurements indicate that the complexity of the image computations is linear in the number of variables, quadratic in the number of template directions, and linear in the number of time-discretization steps.

The accuracy of the overapproximation can be increased arbitrarily by choosing smaller time steps and adding more template directions. To attain a given approximation error (in the Hausdorff sense), the number of template directions is worst-case exponential. In case studies, the developers of SpaceEx observe that a linear number of user-specified directions, possibly augmented by a small set of critical directions, often suffices. The LGG and STC algorithms use floating-point computations that do not formally guarantee soundness. SpaceEx has been used to verify continuous and hybrid systems with more than 100 continuous variables.

ToolboxLS: Level Set Methods

Level set methods are a class of algorithms designed for approximating the solution of the Hamilton–Jacobi partial differential equation (PDE) [127], which arises in many fields including optimal control, differential games, and dynamic implicit surfaces. In particular, dynamic implicit surfaces can be used to compute backward reachable sets and tubes for nonlinear, nondeterministic, continuous dynamic systems with control and/or disturbance inputs; in other words, inputs and parameters to

[14] http://spaceex.imag.fr/.

the model can be treated in a worst case and/or best case fashion. The strengths and weaknesses of the Hamilton–Jacobi PDE formulation of reachability are very similar to those of viability theory: it can treat very general dynamics with adversarial inputs and can represent very general sets, but the known computational algorithms require resources that grow exponentially with the number of dimensions (number of variables); for example, in ToolboxLS[15] the level set algorithms run on a Cartesian grid of the state space. The ToolboxLS algorithms also do not guarantee the sign of computational errors, but they deliver higher accuracy for a given resolution than that available from typical sound alternatives.

Because ToolboxLS [128] is designed for dynamic implicit surfaces rather than specifically for reachability, it does not include a specialized verification interface; however, it has a 140-page user manual documenting the software and over twenty complete examples including three reachable-set computations. It has been used primarily for reachability of systems with two to four continuous dimensions, including collision avoidance, quadrotor flips, aerial refueling, automated landing, and glide-path recapture.

Acknowledgements The authors thank the anonymous reviewers of this chapter for helpful feedback, as well as David Henriques and João Martins for their proofreading help. We also thank Andrew Sogokon for suggesting the nice illustration shown in Fig. 14.

References

1. Proceedings of the 27th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2012, Dubrovnik, Croatia, June 25–28, 2012. IEEE (2012)
2. Althoff, M., Krogh, B.: Reachability analysis of nonlinear differential-algebraic systems. IEEE Trans. Autom. Control **59**, 371–383 (2014)
3. Althoff, M., Krogh, B.H., Stursberg, O.: Analyzing reachability of linear dynamic systems with parametric uncertainties. In: Rauh, A., Auer, E. (eds.) Modeling, Design, and Simulation of Systems with Uncertainties. Springer, Heidelberg (2011)
4. Althoff, M.: Reachability analysis of nonlinear systems using conservative polynomialization and non-convex sets. In: Proceedings of the 16th International Conference on Hybrid Systems: Computation and Control, pp. 173–182. ACM, New York (2013)
5. Althoff, M., Krogh, B.H.: Avoiding geometric intersection operations in reachability analysis of hybrid systems. In: Proceedings of the 15th ACM International Conference on Hybrid Systems: Computation and Control, pp. 45–54. ACM, New York (2012)
6. Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T.A., Ho, P.-H., Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: The algorithmic analysis of hybrid systems. Theor. Comput. Sci. **138**, 3–34 (1995)
7. Alur, R.: Formal verification of hybrid systems. In: Chakraborty et al. [41], pp. 273–278
8. Alur, R., Courcoubetis, C., Henzinger, T.A., Ho, P.-H.: Hybrid automata: an algorithmic approach to the specification and verification of hybrid systems. In: Grossman et al. [82], pp. 209–229
9. Alur, R., Dang, T., Ivancic, F.: Counterexample-guided predicate abstraction of hybrid systems. Theor. Comput. Sci. **354**(2), 250–271 (2006)

[15] http://www.cs.ubc.ca/~mitchell/ToolboxLS/.

10. Alur, R., Dang, T., Ivancic, F.: Predicate abstraction for reachability analysis of hybrid systems. ACM Trans. Embed. Comput. Syst. **5**(1), 152–199 (2006)
11. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. **126**(2), 183–235 (1994)
12. Alur, R., Henzinger, T., Lafferriere, G., Pappas, G.J.: Discrete abstractions of hybrid systems. Proc. IEEE **88**(7), 971–984 (2000)
13. Alur, R., Henzinger, T.A., Ho, P.-H.: Automatic symbolic verification of embedded systems. IEEE Trans. Softw. Eng. **22**(3), 181–201 (1996)
14. Alur, R., Henzinger, T.A., Vardi, M.Y.: Parametric real-time reasoning. In: ACM Symposium on Theory of Computing, pp. 592–601 (1993)
15. Alur, R., La Torre, S., Pappas, G.J.: Optimal paths in weighted timed automata. In: Domenica Di Benedetto, M., Sangiovanni-Vincentelli, A.L. (eds.) HSCC. LNCS, vol. 2034, pp. 49–62. Springer, Heidelberg (2001)
16. Alur, R., Pappas, G.J. (eds.): Hybrid Systems: Computation and Control, Proceedings of the 7th International Workshop, HSCC 2004, Philadelphia, PA, USA, March 25–27, 2004. LNCS, vol. 2993. Springer, Heidelberg (2004)
17. Asarin, E., Dang, T.: Abstraction by projection and application to multi-affine systems. In: Alur and Pappas [16], pp. 32–47
18. Asarin, E., Dang, T., Girard, A.: Hybridization methods for the analysis of nonlinear systems. Acta Inform. **43**(7), 451–476 (2007)
19. Asarin, E., Dang, T., Maler, O., Bournez, O.: Approximate reachability analysis of piecewise-linear dynamical systems. In: Proc. HSCC 00: Hybrid Systems—Computation and Control. LNCS, vol. 1790, pp. 20–31. Springer, Heidelberg (2000)
20. Asarin, E., Dang, T., Maler, O., Testylier, R.: Using redundant constraints for refinement. In: Automated Technology for Verification and Analysis, pp. 37–51. Springer, Heidelberg (2010)
21. Asarin, E., Maler, O., Pnueli, A.: Reachability analysis of dynamical systems having piecewise-constant derivatives. Theor. Comput. Sci. **138**(1), 35–65 (1995)
22. Bagnara, R., Hill, P.M., Zaffanella, E.: The Parma Polyhedra Library: toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Sci. Comput. Program. **72**(1–2), 3–21 (2008)
23. Baier, C., Bertrand, N., Bouyer, P., Brihaye, T., Größer, M.: Almost-sure model checking of infinite paths in one-clock timed automata. In: Proc. of LICS, pp. 217–226. IEEE, Piscataway (2008)
24. Balluchi, A., Di Natale, F., Sangiovanni-Vincentelli, A.L., van Schuppen, J.H.: Synthesis for idle speed control of an automotive engine. In: Alur and Pappas [16], pp. 80–94
25. Behrmann, G., Fehnker, A., Hune, T., Larsen, K.G., Pettersson, P., Romijn, J., Vaandrager, F.W.: Minimum-cost reachability for priced timed automata. In: Proc. of HSCC. LNCS, vol. 2034, pp. 147–161. Springer, Heidelberg (2001)
26. Bemporad, A., Bicchi, A., Buttazzo, G. (eds.): Hybrid Systems: Computation and Control, Proceedings of the 10th International Conference, HSCC 2007, Pisa, Italy. LNCS, vol. 4416. Springer, Heidelberg (2007)
27. Bemporad, A., Morari, M.: Verification of hybrid systems via mathematical programming. In: Vaandrager and van Schuppen [178], pp. 31–45
28. Benerecetti, M., Faella, M., Minopoli, S.: Automatic synthesis of switching controllers for linear hybrid systems: safety control. Theor. Comput. Sci. **493**, 116–138 (2013)
29. Bergstra, J.A., Middelburg, C.A.: Process algebra for hybrid systems. Theor. Comput. Sci. **335**(2–3), 215–280 (2005)
30. Berz, M., Makino, K.: Verified integration of ODEs and flows using differential algebraic methods on high-order Taylor models. Reliab. Comput. **4**(4), 361–369 (1998)
31. Bicchi, A., Pallottino, L.: On optimal cooperative conflict resolution for air traffic management systems. IEEE Trans. Intell. Transp. Syst. **1**(4), 221–231 (2000)
32. Blum, L., Cucker, F., Shub, M., Smale, S.: Complexity and Real Computation. Springer, New York (1998)

33. Bouyer, P., Dufourd, C., Fleury, E., Petit, A.: Updatable timed automata. Theor. Comput. Sci. **321**(2–3), 291–345 (2004)
34. Branicky, M.S.: General hybrid dynamical systems: modeling, analysis, and control. In: Alur, R., Henzinger, T.A., Sontag, E.D. (eds.) Hybrid Systems, vol. 1066, pp. 186–200. Springer, Heidelberg (1995)
35. Branicky, M.S., Borkar, V.S., Mitter, S.K.: A unified framework for hybrid control: model and optimal control theory. IEEE Trans. Autom. Control **43**(1), 31–45 (1998)
36. Brihaye, T., Doyen, L., Geeraerts, G., Ouaknine, J., Raskin, J.-F., Worrell, J.: On reachability for hybrid automata over bounded time. In: Proceedings of ICALP 2011: International Colloquium on Automata, Languages and Programming (Part II). LNCS, vol. 6756, pp. 416–427. Springer, Heidelberg (2011)
37. Bu, L., Li, Y., Wang, L., Li, X.: BACH: bounded reachability checker for linear hybrid automata. In: Formal Methods in Computer-Aided Design, 2008. FMCAD'08, pp. 1–4. IEEE, Piscataway (2008)
38. Buchberger, B.: An algorithm for finding the basis elements of the residue class ring of a zero dimensional polynomial ideal. PhD thesis, University of Innsbruck (1965)
39. Carloni, L.P., Passerone, R., Pinto, A., Sangiovanni-Vincentelli, A.L.: Languages and tools for hybrid systems design. Found. Trends Electron. Des. Autom. **1**(1/2), 1–193 (2006)
40. Cassez, F., Larsen, K.G.: The impressive power of stopwatches. In: CONCUR, pp. 138–152 (2000)
41. Chakraborty, S., Jerraya, A., Baruah, S.K., Fischmeister, S. (eds.): Proceedings of the 11th International Conference on Embedded Software, EMSOFT 2011, Part of the Seventh Embedded Systems Week, ESWeek 2011, Taipei, Taiwan, October 9–14, 2011. ACM, New York (2011)
42. Chase, C., Serrano, J., Ramadge, P.J.: Periodicity and chaos from switched flow systems: contrasting examples of discretely controlled continuous systems. IEEE Trans. Autom. Control **38**(1), 70–83 (1993)
43. Chen, X., Ábrahám, E., Sankaranarayanan, S.: Taylor model flowpipe construction for non-linear hybrid systems. In: RTSS, pp. 183–192. IEEE, Piscataway (2012)
44. Chernikova, N.V.: Algorithm for discovering the set of all solutions of a linear programming problem. USSR Comput. Math. Math. Phys. **8**(6), 282–293 (1968)
45. Chutinan, A., Krogh, B.H.: Verification of polyhedral-invariant hybrid automata using polygonal flow pipe approximations. In: Vaandrager and van Schuppen [178], pp. 76–90
46. Clarke, E.M., Fehnker, A., Han, Z., Krogh, B.H., Ouaknine, J., Stursberg, O., Theobald, M.: Abstraction and counterexample-guided refinement in model checking of hybrid systems. Int. J. Found. Comput. Sci. **14**(4), 583–604 (2003)
47. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Proc. of CAV 2000: Computer Aided Verification. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000)
48. Collet, P., Eckmann, J.P.: Iterated Maps on the Interval as Dynamical Systems, vol. 1. Springer, Heidelberg (1980)
49. Collins, G.E.: Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In: Barkhage, H. (ed.) Automata Theory and Formal Languages. LNCS, vol. 33, pp. 134–183. Springer, Heidelberg (1975)
50. Collins, G.E., Hong, H.: Partial cylindrical algebraic decomposition for quantifier elimination. J. Symb. Comput. **12**(3), 299–328 (1991)
51. Collins, P.: Optimal semicomputable approximations to reachable and invariant sets. Theory Comput. Syst. **41**(1), 33–48 (2007)
52. Cuijpers, P.J.L., Reniers, M.A.: Hybrid process algebra. J. Log. Algebraic Program. **62**(2), 191–245 (2005)
53. Damm, W., Hungar, H., Olderog, E.-R.: Verification of cooperating traffic agents. Int. J. Control **79**(5), 395–421 (2006)
54. Dang, T., Testylier, R.: Reachability analysis for polynomial dynamical systems using the Bernstein expansion. Reliab. Comput. **17**(2), 128–152 (2012)

55. Dantzig, G.B., Eaves, B.C.: Fourier-Motzkin elimination and its dual. J. Comb. Theory **14**, 288–297 (1973)
56. Darboux, J.-G.: Mémoire sur les équations différentielles algébriques du premier ordre et du premier degré. Bull. Sci. Math. Astron. **2**(1), 151–200 (1878)
57. De Wulf, M., Doyen, L., Markey, N., Raskin, J.-F.: Robust safety of timed automata. Form. Methods Syst. Des. **33**(1–3), 45–84 (2008)
58. De Wulf, M., Doyen, L., Raskin, J.-F.: Almost ASAP semantics: from timed models to timed implementations. Form. Asp. Comput. **17**(3), 319–341 (2005)
59. Doyen, L.: Robust parametric reachability for timed automata. Inf. Process. Lett. **102**(5), 208–213 (2007)
60. Doyen, L., Henzinger, T.A., Raskin, J.-F.: Automatic rectangular refinement of affine hybrid systems. In: Proc. of FORMATS 2005: Formal Modelling and Analysis of Timed Systems. LNCS, vol. 3829, pp. 144–161. Springer, Heidelberg (2005)
61. Fainekos, G.E., Girard, A., Pappas, G.J.: Temporal logic verification using simulation. In: Formal Modeling and Analysis of Timed Systems, pp. 171–186. Springer, Heidelberg (2006)
62. Fehnker, A., Krogh, B.H.: Hybrid system verification is not a sinecure—the electronic throttle control case study. Int. J. Found. Comput. Sci. **17**(4), 885–902 (2006)
63. Ferrante, J., Rackoff, C.: A decision procedure for the first order theory of real addition with order. SIAM J. Comput. **4**(1), 69–76 (1975)
64. Fränzle, M.: Analysis of hybrid systems: an ounce of realism can save an infinity of states. In: CSL. LNCS, vol. 1683, pp. 126–140. Springer, Heidelberg (1999)
65. Frazzoli, E., Grosu, R. (eds.): Proceedings of the 14th ACM International Conference on Hybrid Systems: Computation and Control, HSCC 2011, Chicago, USA, April 12–14, 2011. ACM, New York (2011)
66. Frehse, G.: Compositional verification of hybrid systems using simulation relations. PhD thesis, Radboud Universiteit Nijmegen (October 2005)
67. Frehse, G.: PHAVer: algorithmic verification of hybrid systems past HyTech. Int. J. Softw. Tools Technol. Transf. **10**(3), 263–279 (2008)
68. Frehse, G., Le Guernic, C., Donzé, A., Ray, R., Lebeltel, O., Ripado, R., Girard, A., Dang, T., Maler, O.: SpaceEx: scalable verification of hybrid systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV. LNCS, vol. 6806. Springer, Heidelberg (2011)
69. Frehse, G., Kateja, R., Le Guernic, C.: Flowpipe approximation and clustering in space-time. In: Proceedings of the 16th International Conference on Hybrid Systems: Computation and Control, pp. 203–212. ACM, New York (2013)
70. Frehse, G., Krogh, B.H., Rutenbar, R.A.: Verifying analog oscillator circuits using forward/backward abstraction refinement. In: Gielen, G.G.E. (ed.) DATE, pp. 257–262. European Design and Automation Association, Leuven (2006)
71. Freund, R.M., Orlin, J.B.: On the complexity of four polyhedral set containment problems. Math. Program. **33**(2), 139–145 (1985)
72. Fulton, N., Mitsch, S., Quesel, J.-D., Völp, M., Platzer, A.: KeYmaera X: an axiomatic tactical theorem prover for hybrid systems. In: Felty, A.P., Middeldorp, A. (eds.) Automated Deduction, CADE-25. LNCS, vol. 9195, pp. 527–538. Springer, Heidelberg (2015)
73. Ghorbal, K., Platzer, A.: Characterizing algebraic invariants by differential radical invariants. In: Ábrahám, E., Havelund, K. (eds.) TACAS. LNCS, vol. 8413, pp. 279–294. Springer, Heidelberg (2014)
74. Ghosh, P.K., Kumar, K.V.: Support function representation of convex bodies, its application in geometric computing, and some related representations. Comput. Vis. Image Underst. **72**(3), 379–403 (1998)
75. Girard, A.: Reachability of uncertain linear systems using zonotopes. In: Morari, M., Thiele, L. (eds.) HSCC. LNCS, vol. 3414, pp. 291–305. Springer, Heidelberg (2005)
76. Girard, A.: Controller synthesis for safety and reachability via approximate bisimulation. Automatica **48**(5), 947–953 (2012)

77. Girard, A., Le Guernic, C., Maler, O.: Efficient computation of reachable sets of linear time-invariant systems with inputs. In: Hespanha, J.P., Tiwari, A. (eds.) HSCC. LNCS, vol. 3927, pp. 257–271. Springer, Heidelberg (2006)
78. Girard, A., Pappas, G.J.: Approximation metrics for discrete and continuous systems. IEEE Trans. Autom. Control **52**(5), 782–798 (2007)
79. Girard, A., Zheng, G.: Verification of safety and liveness properties of metric transition systems. ACM Trans. Embed. Comput. Syst. **11**(S2), 54 (2012)
80. Goebel, R., Sanfelice, R.G., Teel, A.R.: Hybrid dynamical systems. IEEE Control Syst. Mag. **29**(2), 28–93 (2009)
81. Greenstreet, M.R.: Verifying safety properties of differential equations. In: Computer Aided Verification. LNCS, vol. 1102, pp. 277–287. Springer, Heidelberg (1996)
82. Grossman, R.L., Nerode, A., Ravn, A.P., Rischel, H. (eds.): Hybrid Systems. LNCS, vol. 736. Springer, Heidelberg (1993)
83. Le Guernic, C., Girard, A.: Reachability analysis of hybrid systems using support functions. In: Bouajjani, A., Maler, O. (eds.) CAV. LNCS, vol. 5643, pp. 540–554. Springer, Heidelberg (2009)
84. Gulwani, S., Tiwari, A.: Constraint-based approach for analysis of hybrid systems. In: Gupta, A., Malik, S. (eds.) CAV. LNCS, vol. 5123, pp. 190–203. Springer, Heidelberg (2008)
85. Haghverdi, E., Tabuada, P., Pappas, G.J.: Bisimulation relations for dynamical, control, and hybrid systems. Theor. Comput. Sci. **342**(2), 229–261 (2005)
86. Halbwachs, N., Proy, Y.-E., Raymond, P.: Verification of linear hybrid systems by means of convex approximations. In: International Static Analysis Symposium, SAS'94, Namur (1994)
87. Henzinger, T.A., Ho, P.-H., Wong-Toi, H.: HyTech: the next generation. In: Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS '95), p. 56. IEEE, Piscataway (1995)
88. Henzinger, T.A.: Hybrid automata with finite bisimulations. In: ICALP: Automata, Languages, and Programming. LNCS, vol. 944, pp. 324–335. Springer, Heidelberg (1995)
89. Henzinger, T.A., Ho, P.-H., Wong-Toi, H.: HyTech: a model checker for hybrid systems. Softw. Tools Technol. Transf. **1**, 110–122 (1997)
90. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Proceedings of the 29th Annual Symposium on Principles of Programming Languages, pp. 58–70. ACM, New York (2002)
91. Henzinger, T.A.: The theory of hybrid automata. In: Inan, M.K., Kurshan, R.P. (eds.) Verification of Digital and Hybrid Systems. NATO ASI Series F: Computer and Systems Sciences, vol. 170, pp. 265–292. Springer, Heidelberg (2000)
92. Henzinger, T.A., Ho, P.-H.: A note on abstract interpretation strategies for hybrid automata. In: Hybrid Systems II, pp. 252–264. Springer, Heidelberg (1995)
93. Henzinger, T.A., Ho, P.-H., Wong-Toi, H.: Algorithmic analysis of nonlinear hybrid systems. IEEE Trans. Autom. Control **43**, 540–554 (1998)
94. Henzinger, T.A., Kopke, P.W.: State equivalences for rectangular hybrid automata. In: CONCUR: Concurrency Theory. LNCS, vol. 1119, pp. 530–545. Springer, Heidelberg (1996)
95. Henzinger, T.A., Kopke, P.W.: Discrete-time control for rectangular hybrid automata. Theor. Comput. Sci. **221**, 369–392 (1999)
96. Henzinger, T.A., Kopke, P.W., Puri, A., Varaiya, P.: What's decidable about hybrid automata? J. Comput. Syst. Sci. **57**, 94–124 (1998)
97. Henzinger, T.A., Raskin, J.-F.: Robust undecidability of timed and hybrid systems. In: Proc. of HSCC 00: Hybrid Systems—Computation and Control. LNCS, vol. 1790, pp. 145–159. Springer, Heidelberg (2000)
98. Henzinger, T.A., Wong-Toi, H.: Using HyTech to synthesize control parameters for a steam boiler. In: Abrial, J.-R., Börger, E., Langmaack, H. (eds.) Formal Methods for Industrial Applications. LNCS, vol. 1165, pp. 265–282. Springer, Heidelberg (1995)
99. Ho, P.-H., Wong-Toi, H.: Automated analysis of an audio control protocol. In: Wolper, P. (ed.) CAV. LNCS, vol. 939, pp. 381–394. Springer, Heidelberg (1995)

100. Johnson, T.T., Mitra, S.: Parametrized verification of distributed cyber-physical systems: an aircraft landing protocol case study. In: ICCPS, pp. 161–170. IEEE, Piscataway (2012)
101. Jula, H., Kosmatopoulos, E.B., Ioannou, P.A.: Collision avoidance analysis for lane changing and merging. PATH Research Report UCB-ITS-PRR-99-13, Institute of Transportation Studies, University of California, Berkeley (1999)
102. Agung Julius, A., Fainekos, G.E., Anand, M., Lee, I., Pappas, G.J.: Robust test generation and coverage for hybrid systems. In: Hybrid Systems: Computation and Control, pp. 329–342. Springer, Heidelberg (2007)
103. Kim, K.-D., Kumar, P.R.: Cyber-physical systems: a perspective at the centennial. Proc. IEEE **100**, 1287–1308 (2012). Centennial-Issue
104. Kouskoulas, Y., Renshaw, D.W., Platzer, A., Kazanzides, P.: Certifying the safe design of a virtual fixture control algorithm for a surgical robot. In: Belta, C., Ivancic, F. (eds.) HSCC, pp. 263–272. ACM, New York (2013)
105. Kühn, W.: Rigorously computed orbits of dynamical systems without the wrapping effect. Computing **61**(1), 47–67 (1998)
106. Kurzhanskiy, A.A., Varaiya, P.: Ellipsoidal toolbox (ET). In: 45th IEEE Conference on Decision and Control, pp. 1498–1503. IEEE, Piscataway (2006)
107. Kurzhanskiy, A.A., Varaiya, P.: Ellipsoidal techniques for reachability analysis of discrete-time linear systems. IEEE Trans. Autom. Control **52**(1), 26–38 (2007)
108. Lafferriere, G., Pappas, G.J., Sastry, S.: O-minimal hybrid systems. Math. Control Signals Syst. **13**(1), 1–21 (2000)
109. Lafferriere, G., Pappas, G.J., Yovine, S.: Symbolic reachability computation for families of linear vector fields. J. Symb. Comput. **32**(3), 231–253 (2001)
110. Le Guernic, C.: Reachability analysis of hybrid systems with linear continuous dynamics. PhD thesis, Université Grenoble 1, Joseph Fourier (2009)
111. Lee, E.A., Seshia, S.A.: Introduction to Embedded Systems—A Cyber-Physical Systems Approach (2013). Lulu.com
112. Lerda, F., Kapinski, J., Clarke, E.M., Krogh, B.H.: Verification of supervisory control software using state proximity and merging. In: Hybrid Systems: Computation and Control, pp. 344–357. Springer, Heidelberg (2008)
113. Livadas, C., Lygeros, J., Lynch, N.A.: High-level modeling and analysis of TCAS. Proc. IEEE **88**(7), 926–947 (2000)
114. Loos, S.M., Platzer, A., Nistor, L.: Adaptive cruise control: hybrid, distributed, and now formally verified. In: Butler, M., Schulte, W. (eds.) FM. LNCS, vol. 6664, pp. 42–56. Springer, Heidelberg (2011)
115. Loos, S.M., Witmer, D., Steenkiste, P., Platzer, A.: Efficiency analysis of formally verified adaptive cruise controllers. In: Hegyi, A., De Schutter, B. (eds.) ITSC. Springer, Heidelberg (2013)
116. Lotov, A.V., Bushenkov, V.A., Kamenev, G.K.: Interactive Decision Maps. Applied Optimization, vol. 89. Kluwer Academic, Boston (2004)
117. Lunze, J., Lamnabhi-Lagarrigue, F.: Handbook of Hybrid Systems Control: Theory, Tools, Applications. Cambridge University Press, Cambridge (2009)
118. Lynch, N.A., Segala, R., Vaandrager, F.W.: Hybrid I/O automata. Inf. Comput. **185**(1), 105–157 (2003)
119. Maler, O.: Algorithmic verification of continuous and hybrid systems. In: Int. Workshop on Verification of Infinite-State System (Infinity) (2013)
120. Maler, O., Manna, Z., Pnueli, A.: From timed to hybrid systems. In: de Bakker, J.W., Huizing, C., de Roever, W.P., Rozenberg, G. (eds.) REX Workshop, vol. 600, pp. 447–484. Springer, Heidelberg (1991)
121. Maler, O., Pnueli, A. (eds.): Hybrid Systems: Computation and Control, Proceedings of the 6th International Workshop, HSCC 2003, Prague, Czech Republic, April 3–5, 2003. LNCS, vol. 2623. Springer, Heidelberg (2003)
122. Marwedel, P.: Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems. Springer, Heidelberg (2010)

123. Matringe, N., Vieira Moura, A., Rebiha, R.: Generating invariants for non-linear hybrid systems by linear algebraic methods. In: Cousot, R., Martel, M. (eds.) SAS. LNCS, vol. 6337, pp. 373–389. Springer, Heidelberg (2010)
124. Miller, J.S.: Decidability and complexity results for timed automata and semi-linear hybrid automata. In: Proc. of HSCC 00: Hybrid Systems—Computation and Control. LNCS, vol. 1790, pp. 296–309. Springer, Heidelberg (2000)
125. Milner, R.: An algebraic definition of simulation between programs. In: Cooper, D.C. (ed.) Proc. of the 2nd Int. Joint Conference on Artificial Intelligence, London, UK, September 1971. pp. 481–489. William Kaufmann, British Computer Society, London (1971)
126. Milner, R.: A Calculus of Communicating Systems. LNCS, vol. 92. Springer, Heidelberg (1980)
127. Mitchell, I.M., Bayen, A.M., Tomlin, C.J.: A time-dependent Hamilton-Jacobi formulation of reachable sets for continuous dynamic games. IEEE Trans. Autom. Control **50**(7), 947–957 (2005)
128. Mitchell, I.M., Templeton, J.A.: A toolbox of Hamilton-Jacobi solvers for analysis of nondeterministic continuous and hybrid systems. In: Morari, M., Thiele, L. (eds.) Hybrid Systems: Computation and Control. LNCS, vol. 3414, pp. 480–494. Springer, Heidelberg (2005)
129. Mitra, S., Wang, Y., Lynch, N.A., Feron, E.: Safety verification of model helicopter controller using hybrid input/output automata. In: Maler and Pnueli [121], pp. 343–358
130. Mitsch, S., Ghorbal, K., Platzer, A.: On provably safe obstacle avoidance for autonomous robotic ground vehicles. In: Robotics: Science and Systems (2013)
131. Nerode, A., Yakhnis, A.: Modelling hybrid systems as games. In: Proceedings of the 31st IEEE Conference on Decision and Control, vol. 3, pp. 2947–2952 (1992)
132. Nerode, A., Kohn, W.: Models for hybrid systems: automata, topologies, controllability, observability. In: Grossman et al. [82], pp. 317–356
133. Neumaier, A.: The wrapping effect, ellipsoid arithmetic, stability and confidence regions. In: Validation Numerics, pp. 175–190. Springer, Heidelberg (1993)
134. Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: An approach to the description and analysis of hybrid systems. In: Grossman et al. [82], pp. 149–178
135. Platzer, A.: Differential dynamic logic for hybrid systems. J. Autom. Reason. **41**(2), 143–189 (2008)
136. Platzer, A.: Differential-algebraic dynamic logic for differential-algebraic programs. J. Log. Comput. **20**(1), 309–352 (2010)
137. Platzer, A.: Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics. Springer, Heidelberg (2010)
138. Platzer, A.: Quantified differential invariants. In: Frazzoli and Grosu [65], pp. 63–72
139. Platzer, A.: Stochastic differential dynamic logic for stochastic hybrid programs. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE. LNCS, vol. 6803, pp. 431–445. Springer, Heidelberg (2011)
140. Platzer, A.: A complete axiomatization of quantified differential dynamic logic for distributed hybrid systems. Log. Methods Comput. Sci. **8**(4), 1–44 (2012). Special issue for selected papers from CSL'10
141. Platzer, A.: The complete proof theory of hybrid systems. In: LICS [1], pp. 541–550
142. Platzer, A.: A differential operator approach to equational differential invariants. In: Beringer, L., Felty, A. (eds.) ITP. LNCS, vol. 7406, pp. 28–48. Springer, Heidelberg (2012)
143. Platzer, A.: Logics of dynamical systems. In: LICS [1], pp. 13–24
144. Platzer, A.: The structure of differential invariants and differential cut elimination. Log. Methods Comput. Sci. **8**(4), 1–38 (2012)
145. Platzer, A.: A complete uniform substitution calculus for differential dynamic logic. J. Autom. Reason. **59**(2), 219–265 (2017)
146. Platzer, A.: Logical Foundations of Cyber-Physical Systems. Springer, Heidelberg (2018)
147. Platzer, A., Clarke, E.M.: The image computation problem in hybrid systems model checking. In: Bemporad et al. [26], pp. 473–486

148. Platzer, A., Clarke, E.M.: Computing differential invariants of hybrid systems as fixedpoints. Form. Methods Syst. Des. **35**(1), 98–120 (2009)
149. Platzer, A., Clarke, E.M.: Formal verification of curved flight collision avoidance maneuvers: a case study. In: Cavalcanti, A., Dams, D. (eds.) FM. LNCS, vol. 5850, pp. 547–562. Springer, Heidelberg (2009)
150. Platzer, A., Quesel, J.-D.: KeYmaera: a hybrid theorem prover for hybrid systems. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR. LNCS, vol. 5195, pp. 171–178. Springer, Heidelberg (2008)
151. Platzer, A., Quesel, J.-D.: European Train Control System: a case study in formal verification. In: Breitman, K., Cavalcanti, A. (eds.) ICFEM. LNCS, vol. 5885, pp. 246–265. Springer, Heidelberg (2009)
152. Prabhakar, P., Viswanathan, M.: A dynamic algorithm for approximate flow computations. In: Frazzoli and Grosu [65], pp. 133–142
153. Prajna, S., Jadbabaie, A.: Safety verification of hybrid systems using barrier certificates. In: Alur and Pappas [16], pp. 477–492
154. Prajna, S., Jadbabaie, A., Pappas, G.J.: A framework for worst-case and stochastic safety verification using barrier certificates. IEEE Trans. Autom. Control **52**(8), 1415–1429 (2007)
155. Puri, A.: Dynamical properties of timed automata. In: FTRTFT '98. LNCS, vol. 1486, pp. 210–227. Springer, Heidelberg (1998)
156. Puri, A., Varaiya, P.: Decidability of hybrid systems with rectangular differential inclusion. In: Proc. of CAV. LNCS, vol. 818, pp. 95–104. Springer, Heidelberg (1994)
157. Ratschan, S.: Safety verification of non-linear hybrid systems is quasi-semidecidable. In: TAMC 2010: 7th Annual Conference on Theory and Applications of Models of Computation. LNCS, vol. 6108, pp. 397–408. Springer, Heidelberg (2010)
158. Ratschan, S., She, Z.: Safety verification of hybrid systems by constraint propagation-based abstraction refinement. Trans. Embed. Comput. Syst. **6**(1), 8 (2007)
159. Sankaranarayanan, S.: Automatic invariant generation for hybrid systems using ideal fixed points. In: Johansson, K.H., Yi, W. (eds.) HSCC, pp. 221–230. ACM, New York (2010)
160. Sankaranarayanan, S., Dang, T., Ivančić, F.: Symbolic model checking of hybrid systems using template polyhedra. In: Tools and Algorithms for the Construction and Analysis of Systems, pp. 188–202. Springer, Heidelberg (2008)
161. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Constructing invariants for hybrid systems. Form. Methods Syst. Des. **32**(1), 25–55 (2008)
162. Segelken, M.: Abstraction and counterexample-guided construction of ω-automata for model checking of step-discrete linear hybrid models. In: Damm, W., Hermanns, H. (eds.) CAV. LNCS, vol. 4590, pp. 433–448. Springer, Heidelberg (2007)
163. Sokolsky, O., Lee, I., Heimdahl, M.P.E.: Challenges in the regulatory approval of medical cyber-physical systems. In: Chakraborty et al. [41], pp. 227–232
164. Stursberg, O., Fehnker, A., Han, Z., Krogh, B.H.: Verification of a cruise control system using counterexample-guided search. Control Eng. Pract. **12**(10), 1269–1278 (2004)
165. Tabuada, P.: Verification and Control of Hybrid Systems: A Symbolic Approach. Springer, Heidelberg (2009)
166. Tarski, A.: A Decision Method for Elementary Algebra and Geometry, 2nd edn. University of California Press, Berkeley (1951)
167. Tavernini, L.: Differential automata and their discrete simulators. Nonlinear Anal. **11**(6), 665–683 (1987)
168. Tiwari, A.: Approximate reachability for linear systems. In: Maler and Pnueli [121], pp. 514–525
169. Tiwari, A.: Abstractions for hybrid systems. Form. Methods Syst. Des. **32**(1), 57–83 (2008)
170. Tiwari, A.: Generating box invariants. In: Egerstedt, M., Mishra, B. (eds.) HSCC. LNCS, vol. 4981, pp. 658–661. Springer, Heidelberg (2008)
171. Tiwari, A.: Logic in software, dynamical and biological systems. In: LICS, pp. 9–10. IEEE, Piscataway (2011)

172. Tiwari, A., Shankar, N., Rushby, J.M.: Invisible formal methods for embedded control systems. Proc. IEEE **91**(1), 29–39 (2003)
173. Tiwary, H.R.: On the hardness of computing intersection, union and Minkowski sum of polytopes. Discrete Comput. Geom. **40**(3), 469–479 (2008)
174. Tomlin, C., Pappas, G., Košecká, J., Lygeros, J., Sastry, S.: Advanced air traffic automation: a case study in distributed decentralized control. In: Siciliano, B., Valavanis, K. (eds.) Control Problems in Robotics and Automation. Lecture Notes in Control and Information Sciences, vol. 230, pp. 261–295. Springer, Heidelberg (1998)
175. Tomlin, C., Pappas, G.J., Sastry, S.: Conflict resolution for air traffic management: a study in multi-agent hybrid systems. IEEE Trans. Autom. Control **43**(4), 509–521 (1998)
176. Umeno, S., Lynch, N.A.: Proving safety properties of an aircraft landing protocol using I/O automata and the PVS theorem prover: a case study. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM, vol. 4085, pp. 64–80. Springer, Heidelberg (2006)
177. Umeno, S., Lynch, N.A.: Safety verification of an aircraft landing protocol: a refinement approach. In: Bemporad et al. [26], pp. 557–572
178. Vaandrager, F.W., van Schuppen, J.H. (eds.) Hybrid Systems: Computation and Control, Proceedings of the Second International Workshop, HSCC'99, Berg en Dal, The Netherlands, March 29–31, 1999. LNCS, vol. 1569, Springer, Heidelberg (1999)
179. van Beek, D.A., Man, K.L., Reniers, M.A., Rooda, J.E., Schiffelers, R.R.H.: Syntax and consistent equation semantics of hybrid Chi. J. Log. Algebraic Program. **68**(1–2), 129–210 (2006)
180. van Beek, D.A., Reniers, M.A., Schiffelers, R.R.H., Rooda, J.E.: Concrete syntax and semantics of the compositional interchange format for hybrid systems. In: 17th IFAC World Congress (2008)
181. Wong-Toi, H.: Analysis of slope-parametric rectangular automata. In: Hybrid Systems. LNCS, vol. 1567, pp. 390–413. Springer, Heidelberg (1997)
182. Wongpiromsarn, T., Mitra, S., Murray, R.M., Lamperski, A.G.: Periodically controlled hybrid systems. In: Majumdar, R., Tabuada, P. (eds.) HSCC. LNCS, vol. 5469, pp. 396–410. Springer, Heidelberg (2009)

Chapter 31
Symbolic Model Checking in Non-Boolean Domains

Rupak Majumdar and Jean-François Raskin

Abstract We consider symbolic model checking as a general procedure to compute fixed points on general lattices. We show that this view provides a unified approach for formal reasoning about systems that is applicable to many different classes of systems and properties. Our unified view is based on the notion of region algebras together with appropriate generalizations of the modal μ-calculus. We show applications of our general approach to problems in infinite-state verification, reactive synthesis, and the analysis of probabilistic systems.

31.1 Introduction

Symbolic model checking is an algorithmic technique for exploring properties of dynamical systems. In this technique, the state space of the system is encoded by a finite data structure and system behaviors are explored by manipulating the data structure. This data structure, called a *region*, represents (encodings of) system states, and a set of operations on regions, called a *region algebra*, provides the ingredients necessary to manipulate regions in order to determine whether the behaviors of the system satisfy some given properties.

For example, for invariant verification of transition systems, regions represent sets of states of the transition system, and a region algebra can include Boolean operations on regions (such as union, intersection, and negation), a check for emptiness, and a *successor* operator which, given a region, computes the region representing all states reachable from the given region in one transition. This region algebra is then sufficient to answer whether a given region is an invariant of a transition system, for example, by checking that the initial region is included in the given region and that the successor of the given region is again contained within the region.

A common representation of regions is as predicates in some constraint language, e.g., over Booleans, reals, or integers, using logical operations in the constraint lan-

R. Majumdar (✉)
Max Planck Institute for Software Systems, Kaiserslautern, Germany
e-mail: rupak@mpi-sws.org

J.-F. Raskin
Université libre de Bruxelles, Brussels, Belgium

E.M. Clarke et al. (eds.), *Handbook of Model Checking*,
DOI 10.1007/978-3-319-10575-8_31

guage to implement the region algebra. A formula in the constraint language represents a set of states in the system: those states that satisfy the formula. Boolean operations on regions are implemented using Boolean operations on formulas. The emptiness check is performed by checking satisfiability of the formula. Finally, the successor operation, written $Post(S)$, is performed by quantification: a state s' satisfies $Post(S)$ for a target region S iff there is a state $s \in S$ such that there is a transition from s to s', i.e., if s' satisfies the formula $\exists s.S(s) \wedge T(s, s')$, where $T(s, s')$ is a constraint representation of the transition relation.

For example, in hardware verification, regions can be encoded as (quantifier-free) Boolean formulas whose set of satisfying assignments corresponds to a set of latched values, and the above operations are implemented as Boolean formula manipulations (by eliminating the existential quantifier, $Post(S)$ is again a quantifier-free Boolean formula). Using a data structure such as reduced ordered binary decision diagrams (BDDs) [31], operations on regions can be performed efficiently in practice. In fact, symbolic model checking for hardware circuits using BDD-based representations constituted an early, and convincing, success of model checking [33], so much so that "symbolic model checking" is sometimes synonymous with "BDD-based model checking" (see Chaps. 7 and 8).

However, symbolic model checking transcends BDD-based model checking in several directions: in the expressivity of regions (finite and infinite state spaces, more general constraint languages), in the form of the underlying system being verified (transition systems, games), in the expressivity of formalisms (deterministic vs. stochastic), and in the specification mechanism (Boolean vs. quantitative). In this chapter, we bring together a number of results which can all be formalized and studied under the umbrella of symbolic techniques.

The unifying theme in our treatment of symbolic techniques is the use of the modal μ-calculus [71], a logic that adds inductive definitions to basic modal logic. The inductive structure provides a simple iterative scheme to evaluate a μ-calculus formula; thus, there is a direct connection from a property expressed in the μ-calculus and its symbolic evaluation. The (Boolean) μ-calculus holds a central role in the study of temporal logics, automata, and games (see Chap. 26). In this chapter, we show generalizations of the μ-calculus in various domains, and how these generalizations capture symbolic reasoning about many different classes of systems in a uniform way. In particular, we show how symbolic techniques originally devised for verification can be uniformly extended to synthesis (by applying a game-based generalization of the μ-calculus) and to the analysis of stochastic systems (by applying a quantitative generalization of the μ-calculus).

31.2 Transition Systems and Symbolic Verification

31.2.1 Systems: Transition Systems

A *(labelled) transition system*, LTS for short, $\mathcal{S} = (S, M, \delta)$ consists of a set S of states, a set M of moves, and a transition relation $\delta \subseteq S \times M \times S$. For $s, s' \in S$ and

$a \in M$, we write $s \xrightarrow{a} s'$ if $(s, a, s') \in \delta$. In general, the sets S and M need not be finite. A transition system is *finite* if S and M are both finite. For technical reasons, we shall assume that the transition relation is *serial*: for each $s \in S$ and $m \in M$, there is some $s' \in S$ such that $s \xrightarrow{m} s'$.

A *run* $s_0 a_0 s_1 a_1 \ldots$ of a transition system is a finite or infinite sequence of alternating states and moves such that for each $i \geq 0$, we have $s_i \xrightarrow{a_i} s_{i+1}$. A finite or infinite sequence $s_0 s_1 \ldots$ of states is a *trace* of S if there exists a run $s_0 a_0 s_1 a_1 \ldots$. For a state $s \in S$, a *source-s trace* is a trace $s_0 s_1 \ldots$ such that $s_0 = s$. For a finite run $\pi = s_0 a_0 \ldots s_n$, we define $\text{last}(\pi) = s_n$.

31.2.2 Properties and Algorithms: The μ-Calculus

Let $\mathcal{P}$ be a set of atomic proposition symbols, $\mathcal{V}$ a set of variables, and $\mathcal{F}$ a set of function symbols. The formulas of the μ-calculus are defined by the grammar

$$\varphi ::= p \mid \neg p \mid x \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid f(\varphi) \mid \mu x.\varphi \mid \nu x.\varphi$$

for atomic proposition symbols $p \in \mathcal{P}$, variables $x \in \mathcal{V}$, and function symbols $f \in \mathcal{F}$. We assume that for each function $f \in \mathcal{F}$, there is a *dual* $\text{dual}(f) \in \mathcal{F}$ such that $\text{dual}(\text{dual}(f)) = f$. A μ-calculus formula is *closed* if every variable is bound by a μ or ν fixpoint operator.

The semantics of the μ-calculus is given relative to complete value lattices. A *lattice* $L = (E, \preceq)$ consists of a set E of elements and a partial order $\preceq \subseteq E \times E$, such that for every pair of elements $v_1, v_2 \in E$, there is a unique greatest lower bound $v_1 \sqcap v_2$ and a unique least upper bound $v_1 \sqcup v_2$ in E. A lattice is *complete* if any (not necessarily finite) set of elements from E has a greatest lower bound and a least upper bound in E. A *value lattice* is a complete lattice together with a negation operator $\sim$ such that for each $v \in E$, we have $\sim\sim v = v$ and for each $E' \subseteq E$, we have

$$\sim \bigsqcap E' = \bigsqcup \{\sim v \mid v \in E'\}.$$

A μ-calculus interpretation $\mathcal{I} = (\mathbb{L}, [\![\cdot]\!])$ consists of a value lattice $\mathbb{L} = (E, \preceq)$ and an interpretation $[\![\cdot]\!]$ mapping each proposition $p \in \mathcal{P}$ to an interpretation $[\![p]\!] \in E$ and each function $f \in \mathcal{F}$ to a function $[\![f]\!] : E \to E$ such that $[\![\text{dual}(f)]\!](v) = \sim [\![f]\!](\sim v)$ for each $v \in E$. A *variable environment* $e : \mathcal{V} \to E$ is a function mapping each variable in $\mathcal{V}$ to an element of the value lattice. We write $e[x \mapsto v]$ for the variable environment which maps the variable $x \in X$ to v and maps every variable $y \in \mathcal{V} \setminus \{x\}$ to $e(y)$.

Given a μ-calculus interpretation $\mathcal{I} = (\mathbb{L}, [\![\cdot]\!])$ and a variable environment e, the semantics $[\![\varphi]\!]_e^{\mathcal{I}} \in E$ of a μ-calculus formula φ is defined inductively on the

structure of formulas as follows:

$$\begin{aligned}
[\![p]\!]^{\mathcal{I}}_e &= [\![p]\!] \\
[\![\neg p]\!]^{\mathcal{I}}_e &= \sim [\![p]\!]^{\mathcal{I}}_e \\
[\![x]\!]^{\mathcal{I}}_e &= e(x) \\
[\![f(\varphi)]\!]^{\mathcal{I}}_e &= [\![f]\!]([\![\varphi]\!]^{\mathcal{I}}_e) \\
[\![\varphi_1 \vee \varphi_2]\!] &= [\![\varphi_1]\!]^{\mathcal{I}}_e \sqcup [\![\varphi_2]\!]^{\mathcal{I}}_e \\
[\![\varphi_1 \wedge \varphi_2]\!] &= [\![\varphi_1]\!]^{\mathcal{I}}_e \sqcap [\![\varphi_2]\!]^{\mathcal{I}}_e \\
[\![\mu x.\varphi]\!]^{\mathcal{I}}_e &= \sqcap\{v \in E \mid v = [\![\varphi]\!]^{\mathcal{I}}_{e[x \mapsto v]}\} \\
[\![\nu x.\varphi]\!]^{\mathcal{I}}_e &= \sqcup\{v \in E \mid v = [\![\varphi]\!]^{\mathcal{I}}_{e[x \mapsto v]}\}.
\end{aligned}$$

For a closed μ-calculus formula φ, the semantics $[\![\varphi]\!]^{\mathcal{I}}_e$ is independent of the variable environment e, and hence, we simply write $[\![\varphi]\!]^{\mathcal{I}}$. We also omit the superscript $\mathcal{I}$ when it is clear from the context.

Notice that we specify formulas in the μ-calculus in *negation normal form*, where logical negations are only applied to propositions. This is not a restriction. Consider the following rewrites which "push" negations inward:

$$\begin{aligned}
\neg\neg\varphi &= \varphi & \neg(f(\varphi)) &= \mathrm{dual}(f)(\neg\varphi) \\
\neg(\varphi_1 \vee \varphi_2) &= (\neg\varphi_1 \wedge \neg\varphi_2) & \neg(\varphi_1 \wedge \varphi_2) &= (\neg\varphi_1 \vee \neg\varphi_2) \\
\neg(\mu x.\varphi) &= \nu x.\neg(\varphi[\neg x/x]) & \neg(\nu x.\varphi) &= \mu x.\neg(\varphi[\neg x/x])
\end{aligned}$$

where $\varphi[\neg x/x]$ denotes the formula φ in which all occurrences of the variable x have been replaced by $\neg x$. Using the rewrites, for every closed μ-calculus formula φ, it is possible to construct a formula φ' in negation normal form equivalent to φ, that is, such that for every interpretation $\mathcal{I}$ and variable environment e, we have $[\![\varphi]\!]^{\mathcal{I}}_e = \sim [\![\varphi']\!]^{\mathcal{I}}_e$.

An important special case of the μ-calculus is the *Boolean* μ-calculus interpreted over transition systems.

Fix a transition system $\mathcal{S} = (S, M, \delta)$. Let $\mathbb{L}_{\mathbb{B}} = (2^S, \subseteq)$ be the lattice defined by the subsets of S ordered by set inclusion. For a set $Q \subseteq S$, let $\sim Q = S \setminus Q$. We define two lattice transformers Epre and Apre as follows:

$$\mathrm{Epre}(Q) = \left\{ s \in S \;\middle|\; \exists s' \in Q.\exists m \in M.s \xrightarrow{m} s' \right\} \tag{1}$$

$$\mathrm{Apre}(Q) = \left\{ s \in S \;\middle|\; \forall m \in M.\forall s' \in S.s \xrightarrow{m} s' \Rightarrow s' \in Q \right\} \tag{2}$$

It can be seen that $\mathrm{Epre}(Q) = \sim \mathrm{Apre}(\sim Q)$ in the lattice $\mathbb{L}_{\mathbb{B}}$.

The Boolean μ-calculus is obtained by taking $\mathcal{P}_{\mathbb{B}} = 2^S$, i.e., there is one proposition symbol per subset of states, and $\mathcal{F}_{\mathbb{B}} = \{\mathrm{pre}, \mathrm{dpre}\}$ in the definition of the μ-calculus, where $\mathrm{dual}(\mathrm{pre}) = \mathrm{dpre}$. The semantics of the Boolean μ-calculus on $\mathcal{S}$ is given w.r.t. the lattice $\mathbb{L}_{\mathbb{B}}$ with the following interpretation. Each proposition $p \in 2^S$ is mapped to itself, i.e., $[\![p]\!] = p$ (remember that a proposition is a set of states). Each function in $\mathcal{F}_{\mathbb{B}}$ is defined as follows:

$$[\![\mathrm{pre}]\!]_{\mathcal{S}}(X) = \mathrm{Epre}(X) \quad \text{and} \quad [\![\mathrm{dpre}]\!]_{\mathcal{S}}(X) = \mathrm{Apre}(X)$$

We note that the choice of the lattice $\mathbb{L}$ can have a profound influence on the properties of the logic. For example, it is known that the fixpoint alternation hierarchy of the Boolean μ-calculus is strict [29]. However, in interpretations of the μ-calculus over discounted lattices [2], every formula is equivalent to a formula without fixpoint alternations (see also [34]).

31.2.3 Symbolic Model Checking

The definition of the μ-calculus naturally suggests a model-checking algorithm, where each fixpoint operation is computed using successive approximations [33, 54]. In order to extend it to infinite-state systems, we introduce *symbolic region algebras*.

A symbolic region algebra for a transition system $\mathcal{S}$ and the Boolean μ-calculus consists of a (possibly infinite) set Reg of *regions* and an extension function $\ulcorner\cdot\urcorner : \mathsf{Reg} \to 2^S$ mapping each region to a set of states such that the following conditions are satisfied:

1. Reg contains regions True and False with extensions $\ulcorner\mathsf{True}\urcorner = S$ and $\ulcorner\mathsf{False}\urcorner = \emptyset$.
2. Reg is effectively closed under the Boolean operations: for each pair $r, r' \in \mathsf{Reg}$, there are regions $\mathsf{And}(r, r')$, $\mathsf{Or}(r, r')$, and $\mathsf{Diff}(r, r')$ such that $\ulcorner\mathsf{And}(r, r')\urcorner = \ulcorner r\urcorner \cap \ulcorner r'\urcorner$, $\ulcorner\mathsf{Or}(r, r')\urcorner = \ulcorner r\urcorner \cup \ulcorner r'\urcorner$, and $\ulcorner\mathsf{Diff}(r, r')\urcorner = \ulcorner r\urcorner \setminus \ulcorner r'\urcorner$.
3. Reg is effectively closed under the functions $[\![\mathrm{pre}]\!]$, $[\![\mathrm{dpre}]\!]$: for each $r \in \mathsf{Reg}$ and $\mathrm{fun} \in \{\mathrm{pre}, \mathrm{dpre}\}$, there is a region $r' \in \mathsf{Reg}$ such that $\ulcorner r'\urcorner = [\![\mathrm{fun}]\!](\ulcorner r\urcorner)$.
4. There is a computable function $\mathsf{Member} : S \times \mathsf{Reg} \to \mathbb{B}$ such that $\mathsf{Member}(s, r)$ iff $s \in \ulcorner r\urcorner$.
5. There is a computable function $\mathsf{Empty} : \mathsf{Reg} \to \mathbb{B}$ such that $\mathsf{Empty}(r)$ iff $\ulcorner r\urcorner = \emptyset$.

The tuple $\mathcal{R} = (\mathsf{Reg}, \mathsf{And}, \mathsf{Or}, \mathsf{Diff}, \mathsf{Empty}, \mathsf{Member})$ is called the *region algebra* of $\mathcal{S}$. Conceptually, a region algebra separates the semantics of a transition system (states, transitions) from the data structures required to *represent* the transition system.

Using a region algebra, we can define the symbolic semi-algorithm SMC for μ-calculus model checking shown in Algorithm 1. It takes as input a region algebra, a μ-calculus formula, and a variable environment, and returns a region. The termination test $T \subseteq T'$ is equivalent to checking that $\mathsf{Empty}(\mathsf{And}(T, \mathsf{Diff}(\mathsf{True}, T')))$. It is easy to see the following.

Theorem 1 $\ulcorner\mathsf{SMC}(\mathcal{R}, \varphi, e)\urcorner = [\![\varphi]\!]_e$.

The effectiveness assumption on region algebras implies that each step of the algorithm is computable. The computation of fixpoints is guaranteed to terminate for finite-state systems, but not for infinite state systems in general.

Algorithm 1 Symbolic model-checking procedure $\mathsf{SMC}(\mathcal{R}, \varphi, e)$

Require: region algebra $\mathcal{R} = (\mathsf{Reg}, \ulcorner \cdot \urcorner, \mathsf{And}, \mathsf{Or}, \mathsf{Diff}, \mathsf{Empty}, \mathsf{Member})$, μ-calculus formula φ, variable environment e

match φ **with**
  $p \rightarrow$ **return** p
  $\neg p \rightarrow$ **return** $\mathsf{Diff}(\mathsf{True}, p)$
  $x \rightarrow$ **return** $e(x)$
  $\varphi_1 \vee \varphi_2 \rightarrow$ **return** $\mathsf{Or}(\mathsf{SMC}(\mathcal{R}, \varphi_1, e), \mathsf{SMC}(\mathcal{R}, \varphi_2, e))$
  $\varphi_1 \wedge \varphi_2 \rightarrow$ **return** $\mathsf{And}(\mathsf{SMC}(\mathcal{R}, \varphi_1, e), \mathsf{SMC}(\mathcal{R}, \varphi_2, e))$
  $f(\varphi_1) \rightarrow$ **return** $[\![f]\!](\mathsf{SMC}(\mathcal{R}, \varphi_1, e))$
  $\mu x.\varphi_1 \rightarrow$
    $T_0 = \mathsf{False}$
    for $i = 0, 1, 2, \ldots$ **do**
      $T_{i+1} = \mathsf{SMC}(\mathcal{R}, \varphi_1, e[x \rightarrow T_i])$
    until $T_{i+1} \subseteq T_i$
    return T_i
  $\nu x.\varphi_1 \rightarrow$
    $T_0 = \mathsf{True}$
    for $i = 0, 1, 2, \ldots$ **do**
      $T_{i+1} = \mathsf{SMC}(\mathcal{R}, \varphi_1, e[x \rightarrow T_i])$
    until $T_i \subseteq T_{i+1}$
    return T_i

31.2.4 Examples of Properties and Their Verification Algorithms

The Boolean μ-calculus is an expressive logic and subsumes many temporal logics used for the specification of reactive systems [44]. Hence, a model-checking procedure for the Boolean μ-calculus provides verification algorithms for these logics as well.

In the following, we focus on *linear-time properties*, but the μ-calculus can be used to check *branching-time* properties as well. A linear-time property $\Phi \subseteq S^\omega$ is a set of infinite sequences over S. We are especially interested in *ω-regular properties*, i.e., linear-time properties that are expressible using ω-automata [89]. Let us start with some specific examples of linear-time properties. Let $T \subseteq S$, then:

- The *safety property* $\Box T$ consists of all sequences $\pi = s_0 s_1 \ldots$ such that for each $i \geq 0$, we have $s_i \in T$ (i.e., all states along the trace belong to T).
- The *reachability property* $\Diamond T$ consists of all sequences $\pi = s_0 s_1 \ldots$ such that there exists an $i \geq 0$ such that $s_i \in T$ (i.e., some state along the trace belongs to T).
- The *Büchi property* $\Box\Diamond T$ consists of all sequences $\pi = s_0 s_1 \ldots$ such that for each $i \geq 0$ there exists a $j > i$ such that $s_j \in T$ (i.e., states from T occur infinitely often along the trace).

- The *co-Büchi property* $\Diamond\Box T$ consists of all sequences $\pi = s_0 s_1 \ldots$ such that there exists an $i \geq 0$ such that for all $j > i$ we have $s_j \in T$ (i.e., states from T eventually occur forever along the trace).

We can interpret these properties over transition systems in two ways. Let Φ be a property. In the *existential interpretation*, written $\exists^{\mathbb{B}}_{\mathcal{S}}\Phi$, we say that a state $s \in S$ satisfies the property if there exists a source-s trace that satisfies the property. In the *universal interpretation*, written $\forall^{\mathbb{B}}_{\mathcal{S}}\Phi$, we say that a state $s \in S$ satisfies the property if every source-s trace satisfies the property.

The μ-calculus, and the model-checking algorithm in Algorithm 1, provides a symbolic technique to compute $\exists^{\mathbb{B}}_{\mathcal{S}}\Phi$ and $\forall^{\mathbb{B}}_{\mathcal{S}}\Phi$. Consider the reachability property $\Diamond T$ for a set of states T. We claim that the following holds:

$$\exists^{\mathbb{B}}_{\mathcal{S}}\Diamond T = \left[\!\left[\mu x.T \vee \mathrm{pre}(x)\right]\!\right] \tag{3}$$

To see this, consider the computation of the least fixpoint: $T_0 = \mathsf{False}$, $T_{i+1} = T \vee \mathrm{pre}(T_i)$. By induction on i, one can show that T_i consists of all states from which there is a path of at most i steps to some state in T. The fixpoint is obtained as the limit of this sequence, and thus contains all states that have a path to T. Dually,

$$\forall^{\mathbb{B}}_{\mathcal{S}}\Diamond T = \left[\!\left[\mu x.T \vee \mathrm{dpre}(x)\right]\!\right] \tag{4}$$

Similarly, consider the safety property $\Box T$ for a set of states T. We claim

$$\exists^{\mathbb{B}}_{\mathcal{S}}\Box T = \left[\!\left[\nu x.T \wedge \mathrm{pre}(x)\right]\!\right] \tag{5}$$

To see this, consider the computation of the greatest fixpoint: $T_0 = \mathsf{True}$, $T_{i+1} = T \wedge \mathrm{pre}(T_i)$. By induction on i, one can show that T_i consists of all states from which there is a path of at least i steps that always stays within T. The fixpoint is obtained as the limit of this sequence, and thus contains all states from which there is a way to stay in T forever.

The μ-calculus formulas for Büchi and co-Büchi properties are somewhat more complex, and require an alternation of fixpoint operators:

$$\exists^{\mathbb{B}}_{\mathcal{S}}\Box\Diamond T = \left[\!\left[\nu y.\mu x.\big(\mathrm{pre}(x) \vee \big(T \wedge \mathrm{pre}(y)\big)\big)\right]\!\right]$$
$$\exists^{\mathbb{B}}_{\mathcal{S}}\Diamond\Box T = \left[\!\left[\mu x.\nu y.\big(\mathrm{pre}(x) \vee \big(T \wedge \mathrm{pre}(y)\big)\big)\right]\!\right]$$
$$\forall^{\mathbb{B}}_{\mathcal{S}}\Box\Diamond T = \left[\!\left[\nu y.\mu x.\big(\mathrm{dpre}(x) \vee \big(T \wedge \mathrm{dpre}(y)\big)\big)\right]\!\right]$$
$$\forall^{\mathbb{B}}_{\mathcal{S}}\Diamond\Box T = \left[\!\left[\mu x.\nu y.\big(\mathrm{dpre}(x) \vee \big(T \wedge \mathrm{dpre}(y)\big)\big)\right]\!\right]$$

More generally, one can show that all formulas in linear temporal logic [83] can be systematically translated to the μ-calculus [44]. Briefly, the construction computes a Büchi automaton that accepts exactly the models of the formula [92], and then checks that the set of traces of the system is included in the language of the Büchi automaton or that the set of traces of the system does not intersect

the language of the automaton. These properties can be checked by evaluating a property of the form $\Box\Diamond T$ on the product of the system and the automaton.

The symbolic model-checking algorithms proposed here proceed by *backward* traversal of the state space using the pre operator. While this approach is natural when specifications use future modalities, one can consider a dual region algebra that proceeds by *forward* traversal of the state space, using a post operator that returns the region of states that can be reached from a region in one step. Indeed, in some cases, a forward traversal of the state space may be more efficient because it focuses on the reachable, and hence relevant, portions of the state space. A μ-calculus based on post operations and corresponding symbolic forward algorithms for linear temporal logic specifications are explored in [64].

31.2.5 Equivalence Relations and Termination

As we will see in Sect. 31.3, *effective region algebras* can be defined for a large set of diverse classes of transition systems. Nevertheless, the existence of an effective region algebra for a class of transition systems is not sufficient to ensure the termination of the symbolic procedure given in Algorithm 1. In general, the termination of this symbolic procedure cannot be ensured, as several problems that can be easily expressed in the μ-calculus are undecidable on various classes of transition systems for which effective region algebras exist. Indeed, consider for example counter machines. It is easy to see that given a Presburger definable set, the set of predecessors is also Presburger definable. So, Presburger formulas (for which satisfiability is decidable and Boolean closure is trivial) form an effective region algebra for counter machines [32]. It is well known that the halting problem for counter machines with two counters is undecidable, and so the termination of our symbolic algorithm cannot be ensured for this class.

In [66], several notions of pre-orders and associated equivalence relations are used to prove the existence of finite quotients that ensure termination of the symbolic procedure of Algorithm 1, or termination of symbolic procedures corresponding to fragments of the μ-calculus. We concentrate here on labelled transition systems; more general results about two-player game structures (that will be introduced later in this chapter) can be found in [8].

Let $\mathcal{S} = (S, M, \delta)$ be a labelled transition system, let $\mathcal{P}$ be a set of atomic propositions over the states of $\mathcal{S}$, and $\mathcal{R} = (\mathsf{Reg}, \mathsf{And}, \mathsf{Or}, \mathsf{Diff}, \mathsf{Empty}, \mathsf{Member})$ be an effective region algebra for $\mathcal{S}$ compatible with $\mathcal{P}$, i.e., for each proposition $p \in \mathcal{P}$, there exists a region $r_p \in \mathsf{Reg}$ such that $[\![p]\!] = \ulcorner r_p \urcorner$ (in the sequel we identify the region r_p with p and so we write p for r_p). A binary relation $\preceq \subseteq S \times S$ is a *simulation*[1] on $\mathcal{S}$ with set of propositions $\mathcal{P}$ if for all $s_1, s_2 \in S$, $s_1 \preceq s_2$ implies:

[1] Note that our notion of simulation does not constrain the choice of labels. This is not necessary as we consider a version of the μ-calculus that does not allow constraints on those labels to be expressed.

- for each proposition $p \in \mathcal{P}$, we have that $s_1 \in \ulcorner p \urcorner$ iff $s_2 \in \ulcorner p \urcorner$;
- for each move m_1 and state s_1' such that $(s_1, m_1, s_1') \in \delta$, there exists a move m_2 and state s_2' such that $(s_2, m_2, s_2') \in \delta$, and $s_1' \preceq s_2'$.

Two states $s_1, s_2 \in S$ are *bisimilar*, denoted $s_1 \cong_1^{\mathcal{S}} s_2$, if there is a symmetric simulation $\preceq$ on $\mathcal{S}$ such that $s_1 \preceq s_2$. The state equivalence $\cong_1^{\mathcal{S}}$ is called *bisimilarity*. We say that $\mathcal{S}$ has a *finite bisimilarity quotient* if there exists a bisimilarity $\cong_1^{\mathcal{S}}$ for $\mathcal{S}$ with a finite number of equivalence classes.

Theorem 2 ([66]) *For every (labelled) transition system $\mathcal{S} = (S, M, \delta)$, set of propositions $\mathcal{P}$ and effective region algebra $\mathcal{R}$ compatible with $\mathcal{P}$ such that $\mathcal{S}$ has a finite bisimilarity quotient, the symbolic model-checking algorithm of Algorithm 1 terminates for all the formulas of the μ-calculus.*

Weakenings of the notion of bisimilarity can be used to define coarser equivalence relations on the state spaces of (labelled) transition systems. For example, two states $s_1, s_2 \in S$ are *similar*, noted $s_1 \cong_2^{\mathcal{S}}$, if there exist two simulation relations $\preceq_1$ and $\preceq_2$ such that $s_1 \preceq_1 s_2$ and $s_2 \preceq_2 s_1$. It is easy to see that bisimilarity implies similarity. We say that a transition system $\mathcal{S} = (S, M, \delta)$ has a *finite similarity quotient* if there exists a similarity relation over its state space with a finite number of equivalence classes. Termination of the symbolic model-checking algorithm can be ensured for a fragment of the μ-calculus over labelled transition systems with finite similarity quotients: let $L_1(\mathcal{P})$ be the subset of the μ-calculus where negation is not allowed and the only function considered is the pre function. For that fragment of the μ-calculus, we can state the following termination result:

Theorem 3 ([66]) *For every (labelled) transition system $\mathcal{S} = (S, M, \delta)$, set of propositions $\mathcal{P}$, and effective region algebra $\mathcal{R}$ compatible with $\mathcal{P}$ such that $\mathcal{S}$ has a finite similarity quotient, the symbolic model-checking procedure of Algorithm 1 terminates for the fragment $L_1(\mathcal{P})$ of the μ-calculus.*

31.3 Examples of Symbolic Verification

31.3.1 Program Verification

Consider a simple guarded-command language [50], where a program consists of a set of input variables I, a disjoint set of program variables X, and a set G of *guarded commands* of the form

$$g \mapsto \wedge\{x := e_x \mid x \in X\},$$

where g is a predicate with free variables from $I \cup X$ and $\wedge\{x := e_x \mid x \in X\}$ is a set of simultaneous assignments where each $x := e_x$ is an assignment of the expression e_x (in an unspecified expression syntax) with variables from $I \cup X$ to the variable x.

For a set X of variables, a valuation over X is a mapping from variables in X to values in their respective domains. We denote by $\mathcal{V}_X$ the set of all possible valuations of the variables in X. For a valuation $v \in \mathcal{V}_X$ and a set $X' \subseteq X$, we define $v[X']$ as the restriction of v to X'. For valuations v_X and v_Y over disjoint sets X and Y respectively, define $v_X \oplus v_Y$ as the valuation over $X \cup Y$ defined by extending the function to the domain $X \cup Y$ in the natural way. A valuation is extended from variables to expressions and predicates in the natural way; in particular, given an expression e, we denote by $v(e)$ the value that this expression takes when its variables are interpreted using the valuation v, and we write $v \models g$ if the valuation v makes the guard g true.

Labelled Transition System of a Program. A program defines a labelled transition system $\mathcal{S} = (S, M, \delta)$ in the following way. The set of states S is $\mathcal{V}_X$, i.e., the set of all possible valuations of the variables in X. The set of moves M is $G \times V_I$, i.e., the set of all pairs (g, v_I) composed of a guarded command $g \in G$ and a valuation v_I of the input variables I. Finally, the transition relation δ is defined as follows: $(v_X, (g, v_I), v'_X) \in \delta$ with $g \mapsto \wedge\{x := e_x \mid x \in X\}$, if $v_X \oplus v_I \models g$ and $v'_X[X \setminus \{x\}] = v_X[X \setminus \{x\}]$, and $v'_X(x) = v_X(e_x)$. Note that if $I = \emptyset$, this is the usual notion of guarded command programs (e.g., [50]).

Region Algebra for Programs. A region algebra for guarded-command programs is given by the set of formulas in first-order logic with free variables from V. (The set of terms and atomic predicates in the logic depend on the syntax of expressions and predicates in the language.) The constants True and False correspond to the formulas *true* and *false*, and Boolean operations are defined by the corresponding operations in first-order logic. Membership and emptiness reduce to the satisfiability of formulas in the logic (effectiveness of these operations imply the satisfiability problem is decidable). Finally, $\mathrm{pre}(R)$ can be represented as the formula:

$$\exists I.\exists V'.\Bigg(\bigvee_{(g \to \wedge\{v := e_x \mid x \in X\}) \in G} \big(g(I, X) \wedge x' = e_x(I, V)\big) \wedge R\big(V'\big)\Bigg),$$

where we write "$\exists I$" as shorthand for "$\exists i_1 \ldots \exists i_k$" for all the variables $i_1, \ldots, i_k$ of I, and similarly for V'.

In general, even if each symbolic operation is effective, the iterative computation of fixpoints may not terminate. Consequently, a lot of research has focused on techniques to compute abstractions of the fixpoint—see Chaps. 13 and 15.

Finite-State Systems and BDDs. In case I and V range over Booleans, and both predicates and expressions are Boolean expressions, one gets a symbolic region algebra in propositional logic. That is, a region is a propositional formula over V, and the operations pre_1 and dpre_1 are, by quantifier elimination (using the identity $\exists x.\varphi(x) \equiv \varphi(\mathit{false}) \vee \varphi(\mathit{true})$), again propositional formulas over V. In practice, working directly with Boolean formulas is not convenient, and model checkers use data structures such as *binary decision diagrams* (see Chap. 7 or [31]) for representing Boolean formulas. Binary decision diagrams give a canonical representation for

Boolean formulas, and efficient algorithms to perform Boolean operations on formulas. Moreover, because the underlying state space is finite, the iterative computation of fixpoints is guaranteed to terminate.

Symbolic model checking of hardware circuits using binary decision diagrams has been shown to scale to extremely large state spaces [33, 78], and is the basis for many academic and industrial model checkers. We point the reader to Chap. 7 for more examples and discussions.

A Remark About Worst Case Complexity. While symbolic algorithms are empirically successful, they are not necessarily optimal algorithms for a given problem in a complexity-theoretic sense. For example, reachability in transition systems defined by Boolean constraints is PSPACE-complete, but an implementation with BDDs does not guarantee polynomial memory usage (in fact, a BDD-based model checker works in exponential space in the worst case). So, for finite-state systems, symbolic methods are attractive not because they improve on the worst-case complexity but because they often behave well in practice (and do not exhibit their worst-case complexity).

31.3.2 Antichain-Based Algorithms for Finite-State Automata

As a first application of non-BDD symbolic model checking for finite-state systems, we consider a fundamental problem from automata theory: the universality problem for nondeterministic finite-state automata.

A *nondeterministic* finite-state automaton B is a 5-tuple $(\Sigma, Q, Q_0, \Delta, \alpha)$ (NFA for short), where Σ is a non-empty finite set called the *alphabet*, Q is a non-empty finite set of states, $Q_0 \subseteq Q$ is a non-empty set of initial states, $\Delta : Q \times \Sigma \to 2^Q \setminus \emptyset$ is the nondeterministic transition function mapping a state and a letter to the set of possible successor states, and $\alpha \subseteq Q$ is the non-empty set of final states. A *run* of B over a word $w = w_1 w_2 \dots w_n \in \Sigma^*$ is a sequence of states $\rho = q_0 q_1 \dots q_n$ such that $q_0 \in Q_0$, and $q_i \in \Delta(q_{i-1}, w_i)$ for all i s.t. $1 \leq i \leq n$; ρ is *accepting* if $q_n \in \alpha$. The language of B, denoted by $L(B)$, is the set of words $w \in \Sigma^*$ such that B has an accepting run ρ on w.

The *universality problem* for NFA asks, given an automaton B, whether $L(B) = \Sigma^*$. This problem is PSPACE-COMPLETE [88], and it is a special case of the language inclusion problem. The classical algorithm to solve this problem is to make the automaton deterministic, complement it, and test the complement for emptiness. Alternatively, we can solve it using a symbolic algorithm.

Subset Construction as an LTS. Let $B = (\Sigma, Q, Q_0, \Delta, \alpha)$ be an NFA. We define from B the following LTS $\mathcal{S}_B^{\text{univ}} = (S, M, \delta)$ where:

- $S = 2^Q \setminus \{\emptyset\}$,
- $M = \Sigma$,
- and $(s, \sigma, s') \in \delta$ if $s' = \bigcup_{q \in s} \Delta(q, \sigma)$.

This labelled transition system resembles very much the classical subset construction that is used for determinization. However, in our symbolic algorithm we do not explicitly construct this structure but we explore it symbolically.

Let AllNonFinal be a proposition such that $[\![\mathsf{AllNonFinal}]\!] = \{s \in S \mid s \subseteq Q \setminus \alpha\}$. Clearly, B is not universal if and only if the set of states $[\![\mathsf{AllNonFinal}]\!]$ is reachable from $s_0 = Q_0$ in $\mathcal{S}_B^{\mathsf{univ}}$. This latter reachability problem can be solved by checking whether s_0 belongs to the states satisfying the μ-calculus formula:

$$\mu x.\big(\mathsf{AllNonFinal} \vee \mathrm{pre}(x)\big) \tag{6}$$

To provide an efficient symbolic algorithm for the evaluation of this least fixpoint, we need an adequate region algebra with efficient symbolic operations. As we only need to evaluate a least fixpoint built from monotone operators only, we can rely on a weaker notion of region algebra than the one introduced in Sect. 31.2.3. Let $\mathcal{S} = (S, M, \delta)$ be a labelled transition system. A pre-*positive region algebra* for $\mathcal{S}$ is a region algebra that is closed for the pre operator (not necessarily for the other operators), and for the positive Boolean operators And and Or (and not necessarily for Diff). As such an algebra is not necessarily closed under Diff, we need also to provide an effective procedure to decide inclusion between two regions.

Region Algebra and Symbolic Algorithm for Universality. We now define a pre-positive region algebra for $\mathcal{S}_B^{\mathsf{univ}}$. The set Reg is the set of antichains (for set inclusion) of non-empty sets of states in A, i.e.,

$$\mathsf{Reg} = \big\{A \subseteq S \;\big|\; \forall s, s' \in A.s \subseteq s' \rightarrow s = s'\big\}.$$

The function $\ulcorner\cdot\urcorner : \mathsf{Reg} \rightarrow 2^S$ maps each antichain $A \in \mathsf{Reg}$ to its $\subseteq$-downward closure in S. Formally, for each $A \in \mathsf{Reg}$, we define $\ulcorner A \urcorner = \{s \in S \mid \exists s' \in A.s \subseteq s'\}$.

The necessary operations on regions are defined as follows. Let $\mathsf{Max}_{\subseteq} : 2^S \rightarrow 2^S$ be defined for each $S' \subseteq S$ as: $\mathsf{Max}_{\subseteq}(S') = \{s \in S' \mid \neg\exists s' \in S'.s \subset s'\}$; this function returns the antichain of maximal elements for $\subseteq$ in S'. Let $A \in \mathsf{Reg}$,

$$\mathrm{pre}_1(A) = \mathsf{Max}_{\subseteq}\big(\cup_{s \in A}\big\{\big\{s' \;\big|\; \exists \sigma \in \Sigma.\Delta\big(s', \sigma\big) \subseteq s\big\}\big\}\big). \tag{7}$$

Let $A_1, A_2 \in \mathsf{Reg}$. We define:

- $\mathsf{Or}(A_1, A_2) = \mathsf{Max}_{\subseteq}(A_1 \cup A_2)$,
- $\mathsf{And}(A_1, A_2) = \mathsf{Max}_{\subseteq}\{s \mid \exists s_1 \in A_1, s_2 \in A_2.s = s_1 \cap s_2\}$,
- $\ulcorner A_1 \urcorner \subseteq \ulcorner A_2 \urcorner$ holds iff $\forall s_1 \in A_1 \exists s_2 \in A_2.s_1 \subseteq s_2$.

Finally, observe that for all $s \in S$, $A \in \mathsf{Reg}$, $\mathsf{Member}(s, A)$ iff $s \in \ulcorner A \urcorner$ iff $\exists s' \in A.s \subseteq s'$, and $[\![\mathsf{AllNonFinal}]\!] = \ulcorner\{Q \setminus \alpha\}\urcorner$. Note that all these operations have polynomial time complexity in the size of the antichain A and the automaton B, while the sets that are symbolically manipulated can be of exponential size. This symbolic algorithm for solving universality of NFA has been proposed in [45].

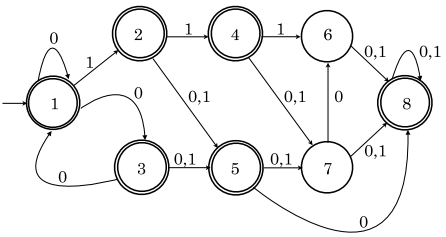


Fig. 1 A finite-state automaton which is not universal

Note that these regions are, by definition, only able to represent $\subseteq$-downward closed sets of sets of states. But this is sufficient in our case as:

- [[AllNonFinal]] is $\subseteq$-downward closed,
- union (and conjunction) of $\subseteq$-downward closed sets are $\subseteq$-downward closed,
- and the pre function maintains $\subseteq$-downward closure.

The intuition for the last point is as follows: if from $s_1 \subseteq Q$ there exists a letter $\sigma \in \Sigma$ such that $s_2 = \delta(s_1, \sigma)$ then, clearly, for all $s_3 \subseteq s_1$, we have that $\delta(s_3, \sigma) \subseteq s_2$. So, if $s_1 \in \text{pre}_1(A)$ then, for all $s_3 \subseteq s_1$, $s_3 \in \text{pre}_1(A)$. As a consequence, all the sets computed during the evaluation of the fixpoint are $\subseteq$-downward closed.

Example 1 Let us consider the finite state automaton depicted in Fig. 1. To check whether this automaton is universal, we evaluate the fixpoint of Eq. (6) that defines the sets of states $s \subseteq Q$ such that there exists a word w for which all runs starting from any state q in s lead to a non-accepting state.

In our example, AllNonFinal is symbolically represented by the antichain $\{\{6, 7\}\}$, and the evaluation of the fixpoint produces the following sequence of antichains:

- $x_0 = \{\{6, 7\}\}$,
- $x_1 = \{\{6, 7\}, \{4, 5\}\}$,
- $x_2 = \{\{6, 7\}, \{4, 5\}, \{2, 3\}\}$,
- $x_3 = \{\{6, 7\}, \{4, 5\}, \{2, 3\}, \{1\}\}$,
- $x_4 = x_3$.

Let us justify the computation of x_1 from x_0. Following Eq. (7), we compute the maximal-$\subseteq$ sets of states $s' \subseteq Q$ for which there exists a letter $\sigma \in \Sigma$ such that all successors of s' by σ are included in $\{6, 7\}$. To compute those sets efficiently, we consider each letter $\sigma \in \Sigma$ and collect all states q of the automaton such that the successors by σ are all in $\{6, 7\}$. In our example, all the successors of 4 by letter 1 are incluced in $\{6, 7\}$ as well as all the 1-successors of 5. As a consequence $\{4, 5\}$ (and any of its subsets) has a letter that leads to a subset of $\{6, 7\}$. For letter 0, the only successor of state 4 is 7. But as $\{4\}$ is included in $\{4, 5\}$, we do not need to add $\{4\}$ explicitly in x_1. All other states have at least one 0-successor and one 1-successor which is not included in $\{6, 7\}$. So, x_1 must represent the union of the sets represented by $\{\{6, 7\}\}$ and $\{\{4, 5\}\}$ only, and so x_1 is equal to the antichain $\{\{6, 7\}, \{4, 5\}\}$. The computations for the other iterations are similar. The fixed point

x_3 allow us to conclude that the automaton of Fig. 1 is not universal as $Q_0 = \{1\} \in \ulcorner x_3 \urcorner$.

Comparison with BDDs. Note that as subsets of Q can be seen as Boolean functions from Q to $\{0, 1\}$, $\mathcal{S}_B^{\mathsf{univ}}$ can be symbolically encoded by BDDs with $|Q|$-Boolean variables for elements of $2^Q \setminus \{\emptyset\}$, and $2|Q|$-Boolean variables for the transition relation. However, it has been shown experimentally that antichain symbolic algorithms often greatly outperform BDD-based symbolic explorations on transition systems defined by variants of the subset construction (which is the case for $\mathcal{S}_B^{\mathsf{univ}}$) [45, 46, 51]. This shows that even for finite systems, alternatives to BDD-based symbolic algorithms may be more efficient.

Antichain solutions have been proposed for other computationally hard problems in automata theory: the language inclusion problem for NFA and the emptiness problem for alternating finite automata on finite words are also solved in [45], the language inclusion problem for nondeterministic Büchi automata and the emptiness problem for alternating Büchi automata are solved in [51]. The universality and language inclusion problems for bottom-up tree automata are solved using antichains in [25].

31.3.3 Timed and Hybrid Systems

As we have seen in the previous section, even if the state space of a system is infinite, symbolic computations can be shown to terminate under special circumstances. This is true, for instance, for timed systems modeled as *timed automata* [11].

Timed Automata. A timed automaton consists of a finite-state automaton together with a set of real-valued clock variables. While the control of the automaton stays within a location, the values of the clocks increase at a constant rate (derivative 1). Associated with each discrete transition between finite locations are a guard condition over clock variables and a subset of clock variables called reset variables. A discrete transition can be taken only when the values of the clock variables satisfy the guard condition, and on executing the transition, each variable in the set of reset variables is set to 0.

Let X be a set of clocks. The set of clock constraints over X, denoted by $\Phi(X)$, is defined by

$$\varphi ::= x \bowtie c \mid x_1 - x_2 \bowtie c \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \neg\varphi$$

where c ranges over positive integer constants, x, x_1, x_2 over clock variables, and $\bowtie \in \{\leq, <, =, >, \geq\}$. A *simple clock constraint* is of the form $x \bowtie c$ or $x_1 - x_2 \bowtie c$. A *convex clock constraint* is a conjunction of simple clock constraints. A *c-clock constraint* is a clock constraint which uses only integer constants smaller than or equal to c. A *valuation* $v : X \to \mathbb{R}_{\geq 0}$ assigns a positive real number to each variable

in X; such a function can equivalently be seen as a vector in $\mathbb{R}_{\geq 0}^{|X|}$. When a valuation v satisfies a clock constraint ϕ, we denote this by $v \models \phi$. Given a valuation v and $t \in \mathbb{R}_{\geq 0}$, we denote by $v + t$ the valuation such that $(v + t)(x) = v(x) + t$ for all $x \in X$. Given a valuation v and $R \subseteq X$, we denote by $v[R := 0]$ the valuation such that $v[R := 0](x) = 0$ if $x \in R$, and $v[R := 0](x) = v(x)$ otherwise.

Formally, a *timed automaton* A is a 5-tuple $(L, X, \Sigma, \mathsf{Inv}, E)$ where L is a non-empty finite set of *locations*, X is a non-empty finite set of *clocks*, Σ is a finite non-empty alphabet, $\mathsf{Inv} : L \to \Phi(X)$ is a function that assigns a convex clock constraint to each location (this constraint is called the *invariant* of the location) and $E \subseteq L \times \Phi(X) \times 2^X \times L$ is a set of transitions of the form (ℓ, ϕ, R, ℓ') where ℓ is the *source location*, ℓ' is the *target location*, ϕ is a *guard*, and R is the set of clocks to be *reset* when the transition is taken.

LTS of a Timed Automaton. We associate with a timed automaton $A = (L, \Sigma, X, \mathsf{Inv}, E)$ a labelled transition system $\mathcal{S}_A = (S, M, \delta)$ defined as follows:

- $S = \{(\ell, v) \in L \times \mathbb{R}_{\geq 0}^{|X|} \mid v \models \mathsf{Inv}(\ell)\}$, *states* are pairs (ℓ, v) where v is a valuation for the clocks in X that satisfies the invariant labeling ℓ;
- $M = E \cup \mathbb{R}_{\geq 0}$, *moves* are either *discrete moves* via transitions or *continuous moves* via time elapsing;
- The transition relation is defined by the following two sub-cases:
  (i) (discrete transitions) $((\ell, v), e, (\ell', v[R := 0])) \in \delta$ if there exists $e = (\ell, \phi, R, \ell') \in E$ such that $v \models \phi$ and $v[R := 0] \models \mathsf{Inv}(\ell')$;
  (ii) (time elapsing transitions) $((\ell, v), t, (\ell, v + t)) \in \delta$ if for all $t' \in [0, t]$, we have $v + t' \models \mathsf{Inv}(\ell)$.

A Region Algebra for Timed Automata. The classical region algebra for timed automata is built on regions that are functions $F : L \to \Phi(X)$. Such functions assign clock constraints with free variables in X to locations. Given a region we define $\ulcorner F \urcorner = \{(\ell, v) \in S \mid v \models F(\ell)\}$. As clock constraints are syntactically closed under Boolean operations, effective operations $\mathsf{And}(\cdot, \cdot)$, $\mathsf{Or}(\cdot, \cdot)$, and $\mathsf{Diff}(\cdot, \cdot)$ exist. Clock constraints are also effectively closed under existential quantification [11], and under universal quantification (as they are closed under complement).

Let F be a region. We construct the region $\mathrm{pre}(F)$, for each $\ell \in L$, by eliminating quantifiers in the formula $\Psi_\ell^E(X) \vee \Psi_\ell^t(X)$, where the formula $\Psi_\ell^E(X)$ represents all the predecessors of F by a discrete transition that leaves ℓ and $\Psi_\ell^t(X)$ represents all the predecessors of F by a time-elapsing transition in ℓ. Ψ_ℓ^E is constructed as follows: for each $e = (\ell, \phi, R, \ell') \in E$, $\Psi_\ell^E(X)$ contains the disjunct $\psi_\ell^e(X)$ defined as:

$$\exists X'.F\big(\ell'\big)\big(X'\big) \wedge \phi(X) \wedge \mathsf{Inv}(\ell, X) \wedge \bigwedge_{x \in X \setminus R} x' = x \wedge \bigwedge_{x \in R} x' = 0 \wedge \mathsf{Inv}\big(\ell', X'\big).$$

Ψ_ℓ^t is constructed as follows:

$$\exists t \geq 0 \exists X'.\mathsf{Inv}(\ell)(X) \wedge \bigwedge_{x \in X} x' = x + t \wedge F(\ell)\big(X'\big) \wedge \mathsf{Inv}(\ell)\big(X'\big).$$

It has also been shown in Chap. 29 that valuations which cannot be distinguished by clock constraints with constants smaller than c are time-abstract bisimilar for any timed automaton in which constraints are c-clock constraints. Furthermore, the number of equivalence classes for c-clock constraints is finite. This guarantees the existence of a finite quotient, known as the region automaton, which is time-abstract bisimilar to the LTS S_A. By Theorem 2, this ensures that the symbolic procedure of Algorithm 1 always terminates for the class of timed automata. In Chap. 29, it is shown that instead of using a region algebra based on clock regions, we can use a more compact representation based on zones [74]. While replacing regions by zones leads to a terminating algorithm for the entire μ-calculus, termination is more difficult to obtain for forward symbolic algorithms using zones, see [28] for details.

Rectangular Hybrid Automata. *Rectangular hybrid automata* extend timed automata by allowing richer dynamics for the continuous variables. Instead of having all the continuous variables evolve with derivative 1 (clocks), rectangular automata allow the use of *flow constraints* of the form $\dot{x} \in I$ where I belongs to the set of *closed intervals* $\mathcal{I} = \{[a, b] \mid a, b \in \mathbb{Q} \cup \{-\infty, +\infty\}\}$. Likewise, the set of guards and updates of variables on transitions must be *rectangular constraints*. Formally, a *rectangular automaton* is a 6-tuple $A = (L, \Sigma, X, \mathsf{Inv}, \mathsf{Flow}, E)$ where L is a finite non-empty set of *locations*, Σ is a finite non-empty *alphabet*, X is a finite non-empty set of *continuous variables*, $\mathsf{Inv} : L \times X \to \mathcal{I}$ assigns to each location ℓ and variable x an interval of values $\mathsf{Inv}(\ell, x)$ which is the *invariant* for the variable x when the control is in location ℓ, $\mathsf{Flow} : L \times X \to \mathcal{I}$ defines for each location ℓ and variable x a *rectangular flow constraint*, and $E \subseteq L \times (X \to \mathcal{I}) \times (X \to \mathcal{I} \cup \mathsf{Id}) \times L$ is a set of *transitions* of the form (ℓ, G, H, ℓ') where ℓ is the *source location*, G assigns a *rectangular guard* to each variable, H specifies the interval of values that can be used to *update* the variable (or Id if the variable is left unchanged by the transition) and ℓ' is the *target location*. We associate with each rectangular automaton $A = (L, \Sigma, X, \mathsf{Inv}, \mathsf{Flow}, E)$ a labelled transition system $\mathcal{S}_A = (S, M, \delta)$:

- $S = \{(\ell, v) \in L \times \mathbb{R}^{|X|}_{\geq 0} \mid v \models \mathsf{Inv}(\ell)\}$,
- $M = E \cup (\mathbb{R}_{\geq 0} \times \mathbb{R}^{|X|})$, i.e., moves are either *discrete moves* via transitions or *continuous moves* via time elapsing together with real-valued flows;
- The transition relation is defined by the two following sub-cases:
  - (i) (discrete transitions) $((\ell, v), e, (\ell', v')) \in \delta$ if there exists $e = (\ell, G, H, \ell') \in E$ such that $v(x) \in G(x)$ for all $x \in X$, either $v'(x) \in H(x)$ or $v'(x) = v(x)$ if $H(x) = \mathsf{Id}$, for all $x \in X$, and $v' \models \mathsf{Inv}(\ell')$,
  - (ii) (flow transitions) $((\ell, v), t, (\ell, v')) \in \delta$ if there exists $(t, d) \in \mathbb{R}_{\geq 0} \times \mathbb{R}^X$ such that $d(x) \in \mathsf{Flow}(\ell, x)$, and for all $t' \in [0, t]$, we have $v + (t' \cdot d) \models \mathsf{Inv}(\ell)$, and for all variables $x \in X$, $v'(x) = v(x) + t \cdot d(x)$, i.e., the values of the continuous variables are evolving while respecting the flow constraints and the location invariant. Here, we write $t \cdot d$ for the function (viewed, equivalently, as a vector) that maps x to $t \cdot d(x)$.

Region Algebra for Rectangular Hybrid Automata. For rectangular hybrid automata, clock constraints are not sufficient to build a region algebra, we need a richer lan-

guage. The language that we use is the first-order logic of the reals $(\mathbb{R}, 0, 1, +, \leq)$ with the reals as domain of interpretation, the constants 0 and 1, the usual order $\leq$ and addition $+$. The satisfiability problem for this logic is decidable [16] and the logic admits effective quantifier elimination. The regions of our algebra are functions F from the set of locations L to quantifier-free formulas of $(\mathbb{R}, 0, 1, +, \leq)$ with free variables in X. As $(\mathbb{R}, 0, 1, +, \leq)$ is closed under all Boolean operations, our region algebra is trivially closed for $\mathsf{And}(\cdot, \cdot)$, $\mathsf{Or}(\cdot, \cdot)$ and $\mathsf{Diff}(\cdot, \cdot)$, and by the decidability of the logic we have effective procedures for $\mathsf{Member}(\cdot, \cdot)$ and $\mathsf{Empty}(\cdot)$. As a consequence we concentrate on the pre operator. Let F be a region. We construct the region $\mathrm{pre}(F)$, for each $\ell \in L$, by eliminating quantifiers in the formula $\Psi_\ell^E(X) \vee \Psi_\ell^t(X)$, where the formula $\Psi_\ell^E(X)$ represents all the predecessors of F by a discrete transition that leaves ℓ, and $\Psi_\ell^t(X)$ represents all the predecessors of F by a flow transition in ℓ. Formally, Ψ_ℓ^E is constructed as follows. For each $e = (\ell, G, H, \ell') \in E$, $\Psi_\ell^E(X)$ contains the conjunct $\psi_\ell^e(X)$:

$$\exists X'. \begin{array}{l} F(\ell')(X') \\ \wedge \bigwedge_{x \in X} x \in G(x) \\ \wedge \bigwedge_{x \in X \mid H(x) = \mathsf{Id}} x = x' \\ \wedge \bigwedge_{x \in X \mid H(x) \neq \mathsf{Id}} x' \in H(x) \\ \wedge \mathsf{Inv}(\ell)(X) \\ \wedge \mathsf{Inv}(\ell')(X'). \end{array} \tag{8}$$

Ψ_ℓ^t is constructed as follows:

$$\exists t \geq 0. \exists X'. \begin{array}{l} F(\ell)(X') \\ \wedge \bigwedge_{x \in X} x + t \cdot \min(\mathsf{Flow}(\ell, x)) \leq x' \leq x + t \cdot \max(\mathsf{Flow}(\ell, x)) \\ \wedge \mathsf{Inv}(\ell)(X) \\ \wedge \mathsf{Inv}(\ell')(X'). \end{array} \tag{9}$$

In practice formulas in $(\mathbb{R}, 0, 1, +, \leq)$ can be efficiently handled as finite unions of convex polyhedra for which there exist efficient implementations, see [14] for example. This has been implemented in tools like HyTech [62] and PHAVer [59]. Unfortunately, termination of the symbolic model-checking algorithm is not ensured anymore for the class of rectangular automata. As shown in Chap. 30, termination of our symbolic algorithm is ensured only for the subclass of *initialized rectangular automata*, see also [65] for the details.

Example 2 We illustrate the use of the logic of the reals as a region algebra for rectangular automata using the example depicted in Fig. 2. The rectangular automaton has two locations ℓ_1 and ℓ_2 and two continuous variables x and y. If we instantiate the formula of Eq. (9) to express the set of states that can reach, by letting time elapse, a state from which the transition from ℓ_1 to ℓ_2 can be taken, we get the

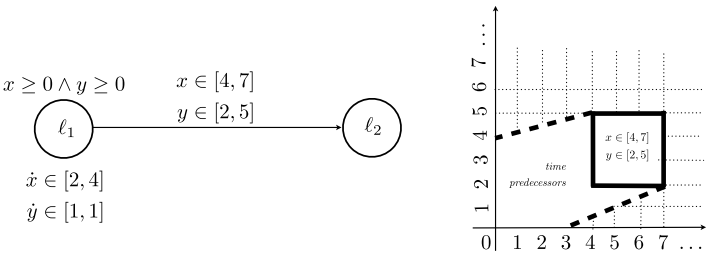


Fig. 2 A simple rectangular automaton and the time predecessors for the guard of the transition from ℓ_1 to ℓ_2

following formula:

$$\exists x'.\exists y'.\exists t \geq 0. \begin{array}{l} x' \in [4,7] \wedge y' \in [2,5] \\ \wedge\ x + t \cdot 2 \leq x' \leq x + t \cdot 4 \wedge y + t \leq y' \leq y + t \\ \wedge\ x \geq 0 \wedge y \geq 0 \\ \wedge\ x' \geq 0 \wedge y' \geq 0. \end{array}$$

By eliminating the existentially quantified variables, we obtain the following description of these states:

$$x \leq 7 \wedge y \leq 5 \wedge y \leq \frac{x}{4} + 4 \wedge y \geq \frac{x}{2} - \frac{3}{2}.$$

31.3.4 Well-Structured Transition Systems

We now turn to a general class of infinite-state transition systems for which a symbolic reachability algorithm terminates: the well-structured transition systems [1, 58], WSTS for short. Well-quasi-order is the basic concept that underlies the definition of WSTS. A *well-quasi-order*, wqo for short, $\leq \subseteq S \times S$ on a set S is a pre-order (a reflexive and transitive relation) such that for all infinite sequences $s_0 s_1 \ldots s_n \cdots \in S^\omega$, there always exist two positions $i < j$ such that $s_i \leq s_j$. We call such a pair $(S, \leq)$ a *well-quasi-ordered set*. A set $U \subseteq S$ is $\leq$*-upward closed* if for all $s_1, s_2 \in S$ if $s_1 \leq s_2$ and $s_1 \in U$ then $s_2 \in U$. Well-quasi-ordered sets enjoy the following properties:

Lemma 1 *Let* $(S, \leq)$ *be a well-quasi-ordered set*:

- *any antichain of elements in* S *is finite*,
- *for all chains* $U_0 \subseteq U_1 \subseteq \cdots \subseteq U_n \subseteq \ldots$ *of* $\leq$*-upward closed subsets of* S, *there exists* $i \geq 0$ *such that for all* $j \geq i$, $U_i = U_j$.

To ease the formalization, for the rest of this section we make the hypothesis that the well-quasi-orders that we consider are asymmetric, i.e., they are partial orders,[2] and we refer to these partially ordered sets as *well-partially-ordered sets*.

Let U be an $\leq$-upward closed set, and define $\mathsf{Min}_{\leq}(U) = \{s \in U \mid \forall s' \in U.s' \leq s \rightarrow s = s'\}$. Clearly, if $\leq$ is a partial order, then $\mathsf{Min}_{\leq}(U)$ is an antichain and so it is finite; furthermore it canonically represents U in the sense that $U = \{s \in S \mid \exists s' \in \mathsf{Min}(U).s' \leq s\}$. We are now ready to define the notion of well-structured transition systems, and show how, using adequate region algebras, we can obtain terminating symbolic algorithms for reachability problems.

A labelled transition system $\mathcal{S} = (S, M, \delta)$ is a *well-structured (labelled) transition system*, WSTS for short, if there exists a relation $\leq \subseteq S \times S$ such that:

- $\leq$ is a well-partial-order,
- for all $s_1, s_2, s_3 \in S$ for all $m \in M$, if $(s_1, m, s_2) \in \delta$ and $s_1 \leq s_3$ then there exist $s_4 \in S$ and $m' \in M$ such that $(s_3, m', s_4) \in \delta$ and $s_2 \leq s_4$.

For WSTS, we consider the *coverability problem* which, given $\mathcal{S}$, s_0, and an $\leq$-upward closed set $U \subseteq S$, asks whether there exists a path in $\mathcal{S}$ from s_0 to some element in U. We can reduce the coverability problem to the problem of checking whether s_0 belongs to

$$[\![\mu x.r_U \vee pre_1(x)]\!] \tag{10}$$

which can be done symbolically if there exists a pre-positive region algebra for $\mathcal{S}$ where r_U is a region such that $\ulcorner r_U \urcorner = U$. The termination of the fixpoint evaluation for this formula is ensured by the fact that successive approximations of the fixpoint form an ascending chain of $\leq$-upward closed sets and by the second property in Lemma 1. We illustrate this on Petri nets [84] which are well known examples of WSTS [1, 58].

Petri Nets as an Example of WSTS. A Petri net N is a 4-tuple (P, T, I, O) where P is a finite set of places, T is a finite set of transitions, $I : P \times T \rightarrow \mathbb{N}$ and $O : T \times P \rightarrow \mathbb{N}$ are the input and output functions. A marking for the Petri net N is a function $s : P \rightarrow \mathbb{N}$, or equivalently a vector in $\mathbb{N}^{|P|}$. A transition t is *enabled* in a marking s if $s(p) \geq I(p, t)$ for all $p \in P$. If a transition t is enabled in a marking s, then from s, t leads to the marking s' such that

$$s'(p) = s(p) - I(p, t) + O(t, p). \tag{11}$$

According to this semantics, we map a Petri net N to a labelled transition system $\mathcal{S}_N = (S, M, \delta)$ where

- $S = \mathbb{N}^{|P|}$,
- $M = T$,

[2] This is not a strong restriction, as every pre-ordered set can be transformed into a partially ordered set by considering the induced partial order on the equivalence classes defined by the pre-order.

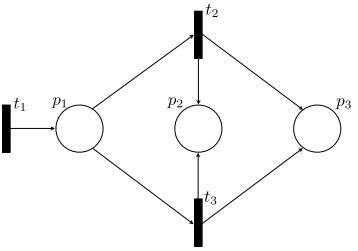


Fig. 3 A Petri net modeling a critical section (p_3) for any number of processes

- and $(s, t, s') \in \delta$ if $s(p) \geq I(p, t)$ and $s'(p) = s(p) - I(p, t) + O(t, p)$ for all $p \in P$.

With $\mathcal{S}_N$, we associate the following pre-positive region algebra.

Region Algebra for Petri Nets. The set $(\mathbb{N}^{|P|}, \leq)$, where $\leq$ is the extension of the ordering of natural number to tuples, is a well-partial-order [49]. The set of regions Reg is the set of antichains of vectors in $\mathbb{N}^{|P|}$. As a consequence, each $A \in \mathsf{Reg}$ is a finite set of vectors in $\mathbb{N}^{|P|}$. The function $\ulcorner \cdot \urcorner : \mathsf{Reg} \to 2^S$ is defined as follows: for each $A \in \mathsf{Reg}$, $\ulcorner A \urcorner = \{s \in S \mid \exists s' \in A. s' \leq s\}$, i.e., each antichain $A \in \mathsf{Reg}$ represent its $\leq$-upward closure in S.

We now define the necessary operations on this set of regions. For this we need the function $\mathsf{Min}_{\leq} : 2^S \to 2^S$ defined for each $B \subseteq S$ as: $\mathsf{Min}_{\leq}(B) = \{s \in B \mid \neg \exists s' \in B. s' < s\}$ that returns the antichain of minimal elements for $\leq$ in B. Let $A_1, A_2 \in \mathsf{Reg}$:

- $\mathsf{Or}(A_1, A_2) = \mathsf{Min}_{\leq}(A_1 \cup A_2)$, and
- $\mathsf{And}(A_1, A_2) = \mathsf{Min}_{\leq}\{s \mid \exists s_1 \in A_1, s_2 \in A_2 \forall p \in P. s(p) = \max(s_1(p), s_2(p))\}$.

Let $A \in \mathsf{Reg}$. $\mathsf{pre}(A)$ is defined as:

$$\mathsf{Min}_{\leq}\left[\bigcup_{s \in A} \bigcup_{t \in T} \left\{ s' \;\middle|\; s'(p) = \max_{\leq}\big(I(p, t), s(p) - \big(O(t, p) - I(p, t)\big)\big) \right\}\right] \tag{12}$$

It can be easily verified that $\ulcorner \mathsf{pre}(A) \urcorner = \mathsf{pre}(\ulcorner A \urcorner)$, and that the pre operation on regions is effective. Finally, observe that for all $s \in S$, $A \in \mathsf{Reg}$, $\mathsf{Member}(s, A)$ iff $s \in \ulcorner A \urcorner$ iff $\exists s' \in A. s' \leq s$. This region algebra allows us to decide membership in an upward closed set as $s \in \ulcorner A \urcorner$ by checking that $\exists s' \in A. s' \leq s$, and inclusion between two regions as $\ulcorner A_1 \urcorner \subseteq \ulcorner A_2 \urcorner$ by checking that $\forall s_1 \in A_1 \exists s_2 \in A_2. s_2 \leq s_1$.

Example 3 To illustrate the backward symbolic algorithm based on the antichain representation of upward closed sets of tuples in $\mathbb{N}^k$, let us consider the example depicted in Fig. 3. This simple Petri net models a system where any number of processes can be created (transition t_1), they can enter the critical section (transition t_2) and leave it (transition t_3). Let us now compute the set of markings that can reach a marking where the mutual exclusion is violated for the critical section, that is the set of markings that can reach a marking in $U = \{(x_1, x_2, x_3) \mid x_3 \geq 2\}$. The set U

is symbolically represented by its unique minimal element: $r_U = \{(0, 0, 2)\}$, and the evaluation of the least fixpoint is as follows:

- $r_{U_0} = r_U = \{(0, 0, 2)\}$
- $r_{U_1} = r_{U_0}$ Or $\text{pre}(r_{U_0}) = \{(0, 0, 2), (1, 1, 1)\}$
- $r_{U_2} = r_{U_1}$ Or $\text{pre}(r_{U_1}) = \{(0, 0, 2), (0, 1, 1), (2, 2, 0)\}$
- $r_{U_3} = r_{U_2}$ Or $\text{pre}(r_{U_2}) = \{(0, 0, 2), (0, 1, 1), (1, 2, 0)\}$
- $r_{U_4} = r_{U_3}$ Or $\text{pre}(r_{U_3}) = \{(0, 0, 2), (0, 1, 1), (0, 2, 0)\}$
- $r_{U_5} = r_{U_4}$ Or $\text{pre}(r_{U_4}) = \{(0, 0, 2), (0, 1, 1), (0, 2, 0)\} = r_{U_4}$

The set $\ulcorner r_{U_4} \urcorner$ contains all the markings in which the mutual exclusion is already violated, i.e., $\ulcorner \{\{(0, 0, 2)\}\} \urcorner$, the markings in which there is at least one process in the critical section and at least one token in place p_2 that will allow for another process to enter the critical section, i.e., $\ulcorner \{\{(0, 1, 1)\}\} \urcorner$, and finally, the markings where there are at least two tokens in p_2 that can be used by two processes to be in the critical section together, i.e., $\ulcorner \{\{(0, 2, 0)\}\} \urcorner$.

To illustrate how this sequence of regions is computed, we consider the computation of r_{U_1} from r_{U_0}, which is done by applying Eq. (12). So first, we need to compute for each transition t in $\{t_1, t_2, t_3\}$, an antichain representation of the predecessors of the set of markings in $\ulcorner r_{U_0} \urcorner$ by t. For t_1, this set is represented by $\{(0, 0, 2)\}$, for t_2, it is represented by $\{(1, 1, 1)\}$, and for t_3, it is represented by $\{(0, 0, 3)\}$. Second, we extract the minimal elements from those three antichains and we obtain $\{(0, 0, 2), (1, 1, 1)\}$ as $(0, 0, 3)$ is eliminated because it is subsumed by $(0, 0, 2)$.

31.4 Games and Symbolic Synthesis

We now shift the focus from verification to synthesis. In verification, one starts with a given system and checks whether a property holds. In synthesis, one starts with a partial description of the system and an environment, and a desired property, and constructs a system that is guaranteed to satisfy the property no matter how the environment behaves. One way to model synthesis problems is as two-person games between a system and its environment. We show that the framework of the μ-calculus, and symbolic algorithms, carry over smoothly from verification to synthesis.

31.4.1 Deterministic Games

A *deterministic game structure* $\mathcal{G} = (S, M, \Gamma_1, \Gamma_2, \delta)$ consists of a set S of states, a set M of moves, two functions $\Gamma_1 : S \to 2^M \setminus \{\emptyset\}$ and $\Gamma_2 : S \to 2^M \setminus \{\emptyset\}$ mapping states to non-empty subsets of moves, and a transition function $\delta : S \times M \times M \to S$.

Intuitively, deterministic game structures naturally define two-player games on graphs. At each state s, player 1 picks a move $a_1 \in \Gamma_1(s)$, while simultaneously

and independently player 2 picks a move $a_2 \in \Gamma_2(s)$, and the state is updated to the successor state given by $\delta(s, a_1, a_2)$.

A *run*

$$s_0 \xrightarrow{a_1^0, a_2^0} s_1 \xrightarrow{a_1^1, a_2^1} \dots \tag{13}$$

is a finite or infinite sequence of alternating states and move-pairs such that for each $i \geq 0$, we have $a_j^i \in \Gamma_j(s_i)$ for $j \in \{1, 2\}$ and $\delta(s_i, a_1^i, a_2^i) = s_{i+1}$. A *trace* associated with the run (13) is the sequence $s_0 s_1 \dots$ of states in the run. For a finite run $\pi = s_0 \dots s_n$, we define $\text{last}(\pi) = s_n$.

A (deterministic) *strategy* ξ_i for player $i \in \{1, 2\}$ is a mapping $\xi_i : S^+ \to M$ that associates with each finite run π a move $\xi_i(\pi)$ that is used by player i when the history of the game has produced the trace π. We require that $\xi_i(\pi) \in \Gamma_i(\text{last}(\pi))$, that is, the move suggested by the strategy is available to player i at the current state. The set of all strategies of player i is denoted Ξ_i, for $i \in \{1, 2\}$.

Let $s \in S$ and let $\xi_1 \in \Xi_1$ and $\xi_2 \in \Xi_2$ be strategies of player 1 and player 2 respectively. Together, these define a unique run of the form (13), where $s = s_0$, and for each $i \geq 0$ and $j \in \{1, 2\}$, we have $a_j^i = \xi_j(s_0 \dots s_i)$. The *outcome* $\text{Outcome}(s, \xi_1, \xi_2) \in S^\omega$ is the trace associated with this run.

We also consider special cases of deterministic game structures. A game structure is *turn-based* if for each state $s \in S$, either $|\Gamma_1(s)| = 1$ or $|\Gamma_2(s)| = 1$, that is, at each state, at most one player has a non-trivial choice of moves. A game structure is a player i structure for $i \in \{1, 2\}$ if for each state, we have $|\Gamma_{3-i}(s)| = 1$; that is, if player i is the only player with a non-trivial choice of moves. Player i structures coincide with labelled transition systems.

31.4.2 The Boolean μ-Calculus on Games

Fix a game structure $\mathcal{G} = (S, M, \Gamma_1, \Gamma_2, \delta)$. In the definition of the μ-calculus, let $\mathcal{P}_\mathbb{B}$ be a finite set of propositions and let $\mathcal{F}_\mathbb{B} = \{\text{pre}_1, \text{dpre}_1, \text{pre}_2, \text{dpre}_2\}$, with $\text{dual}(\text{pre}_1) = \text{dpre}_1$ and $\text{dual}(\text{pre}_2) = \text{dpre}_2$. Let $\mathbb{L}_\mathbb{B} = (2^S, \subseteq)$ be the lattice defined by subsets of S ordered according to set inclusion. For a set $S' \subseteq S$, let $\sim S' = S \setminus S'$.

The semantics of the Boolean μ-calculus on $\mathcal{G}$ is given w.r.t. the lattice $\mathbb{L}_\mathbb{B}$ with the following interpretation. Each proposition $p \in \mathcal{P}_\mathbb{B}$ is mapped to a subset of states. Each function in $\mathcal{F}_\mathbb{B}$ is defined as follows:

$$[\![\text{pre}_1]\!]_\mathcal{G}(X) = \left\{ s \in S \;\middle|\; \exists a_1 \in \Gamma_1(s) \forall a_2 \in \Gamma_2(s).\delta(s, a_1, a_2) \in X \right\}$$

$$[\![\text{dpre}_1]\!]_\mathcal{G}(X) = \left\{ s \in S \;\middle|\; \forall a_1 \in \Gamma_1(s) \exists a_2 \in \Gamma_2(s).\delta(s, a_1, a_2) \notin X \right\}$$

$$[\![\text{pre}_2]\!]_\mathcal{G}(X) = \left\{ s \in S \;\middle|\; \exists a_2 \in \Gamma_2(s) \forall a_1 \in \Gamma_1(s).\delta(s, a_1, a_2) \in X \right\}$$

$$[\![\text{dpre}_2]\!]_\mathcal{G}(X) = \left\{ s \in S \;\middle|\; \forall a_2 \in \Gamma_2(s) \exists a_1 \in \Gamma_1(s).\delta(s, a_1, a_2) \notin X \right\}.$$

With the preceding definition, we can extend symbolic region algebras from transition systems to games, by requiring that the set of regions is effectively closed

under the functions $[\![\mathrm{pre}_1]\!]$, $[\![\mathrm{dpre}_1]\!]$, $[\![\mathrm{pre}_2]\!]$, $[\![\mathrm{dpre}_2]\!]$: for each $r \in \mathsf{Reg}$ and $\mathrm{fun} \in \{\mathrm{pre}_1, \mathrm{dpre}_1, \mathrm{pre}_2, \mathrm{dpre}_2\}$, there is a region $r' \in \mathsf{Reg}$ such that $\ulcorner r' \urcorner = [\![\mathrm{fun}]\!](\ulcorner r \urcorner)$. Using this extension, we can use the symbolic semi-algorithm from Algorithm 1 to symbolically check properties on games.

31.4.3 Linear-Time Properties

We focus again on linear-time properties, specifically, properties expressible in linear temporal logic. Unlike verification, solving games requires constructing deterministic ω-automata, and therefore, it is not enough to consider Büchi automata. (It is known that deterministic Büchi automata accept a strict subset of ω-regular languages [89].)

Instead, we go through *deterministic parity automata*. The parity condition is more complex than the Büchi accepting condition. Let $T_1, T_2, \ldots, T_k$ form a partition of S into k subsets. For a trace $\pi = s_0 s_1 \ldots$, define $\mathrm{Index}(\pi; T_1, \ldots, T_k)$ to be the largest $i \in \{1, \ldots, k\}$ such that $s_j \in T_i$ for infinitely many j. Then, $\mathrm{Parity}(T_1, \ldots, T_k)$ is defined as the set of traces with even index, that is, $\mathrm{Parity}(T_1, \ldots, T_k) = \{\pi \in S^\omega \mid \mathrm{Index}(\pi; T_1, \ldots, T_k) \text{ is even}\}$. It is known that any ω-regular property can be accepted by a deterministic automaton with a parity accepting condition [90].

Let $\mathcal{G}$ be a game structure and Φ a linear property. Define

$$\begin{aligned} \langle 1 \rangle^{\mathbb{B}}_{\mathcal{G}} \Phi &= \left\{ s \in S \;\middle|\; \exists \xi_1 \in \Xi_1 \forall \xi_2 \in \Xi_2 . \mathrm{Outcome}(s, \xi_1, \xi_2) \in \Phi \right\} \\ \langle 2 \rangle^{\mathbb{B}}_{\mathcal{G}} \Phi &= \left\{ s \in S \;\middle|\; \exists \xi_2 \in \Xi_2 \forall \xi_1 \in \Xi_1 . \mathrm{Outcome}(s, \xi_1, \xi_2) \in \Phi \right\} \end{aligned} \tag{14}$$

Intuitively, $\langle 1 \rangle^{\mathbb{B}}_{\mathcal{G}} \Phi$ consists of all states in S from which player 1 has a strategy to force the outcome to be in Φ, no matter what player 2 does. We say player 1 can *control for* the property Φ. The predicate $\langle 2 \rangle^{\mathbb{B}}_{\mathcal{G}} \Phi$ is defined analogously.

The connection between ω-regular properties and the Boolean μ-calculus is given by the following result that provides a symbolic algorithm for checking parity games.

Theorem 4 ([53]) *For every game structure $\mathcal{G}$, partition $T_1, \ldots, T_k$ of the states of $\mathcal{G}$, and players $i \in \{1, 2\}$, we have*

$$\langle 1 \rangle^{\mathbb{B}}_{\mathcal{G}} \mathrm{Parity}(T_1, \ldots, T_k) = \left[\!\!\left[\lambda_k x_k \ldots \nu x_2 \mu x_1 . \bigvee_{j=1}^{k} \left(T_i \wedge \mathrm{pre}_i(x_j) \right) \right]\!\!\right] \tag{15}$$

where $\lambda_k = \nu$ if k if even and $\lambda_k = \mu$ if k is odd.

Theorem 4 suggests a direct symbolic implementation for synthesis against ω-regular objectives: construct a deterministic parity automaton for the ω-regular objective, take the product of the automaton and the game, and then evaluate the μ-calculus formula in (15). Notice that the fixpoint computation is exponential in k,

the number of partitions. Whether there is a polynomial-time algorithm to solve parity games is currently open, see Chaps. 26 and 27.

For specific classes of linear properties, it is possible to get polynomial-time algorithms. For example, polynomial-time algorithms are easily obtained for reachability, safety, Büchi, and co-Büchi objectives. For many practical applications of reactive synthesis, one can write objectives in the form:

$$(\Box\Diamond p_1 \wedge \cdots \wedge \Box\Diamond p_m) \Rightarrow (\Box\Diamond q_1 \wedge \cdots \wedge \Box\Diamond q_n)$$

where p_i, q_j are Boolean combinations of atomic propositions. This class is called the generalized reactivity(1) class of formulas (GR(1)) [20]. For this class, [20] showed a symbolic cubic-time algorithm, through the (equational) μ-calculus formula:

$$\begin{aligned}
Z_1 &=^{\nu} \mu Y. \bigvee_{i=1}^{m} \nu X. \big(q_1 \wedge \mathrm{pre}_1(Z_2) \vee \mathrm{pre}_1(Y) \vee \neg p_i \wedge \mathrm{pre}_1(X)\big) \\
Z_2 &=^{\nu} \mu Y. \bigvee_{i=1}^{m} \nu X. \big(q_2 \wedge \mathrm{pre}_1(Z_3) \vee \mathrm{pre}_1(Y) \vee \neg p_i \wedge \mathrm{pre}_1(X)\big) \\
&\cdots \\
Z_n &=^{\nu} \mu Y. \bigvee_{i=1}^{m} \nu X. \big(q_n \wedge \mathrm{pre}_1(Z_1) \vee \mathrm{pre}_1(Y) \vee \neg p_i \wedge \mathrm{pre}_1(X)\big).
\end{aligned}$$

Symbolic algorithms for synthesis for GR(1) objectives based on BDDs have been implemented in several tools [67, 72], and used to synthesize hardware protocols or robot plans.

One might expect a *determinacy* result for games, which states that $\langle 1\rangle^{\mathbb{B}}_{\mathcal{G}} \Phi = S \setminus \langle 2\rangle^{\mathbb{B}}_{\mathcal{G}}(S^{\omega} \setminus \Phi)$, that is, from each state, either player 1 can control for the property Φ or player 2 can control for the complement of Φ. For example, determinacy would imply $\langle 1\rangle^{\mathbb{B}}_{\mathcal{G}} \Diamond T = S \setminus \langle 2\rangle^{\mathbb{B}}_{\mathcal{G}} \Box(S \setminus T)$, that is, from each state, either player 1 can control for $\Diamond T$ (eventually reach T) or player 2 can control for $\Box(S \setminus T)$ (remain out of T forever). Unfortunately, determinacy does not hold for deterministic game structures. It is not the case that $S \setminus \mathrm{pre}_1(S') = \mathrm{pre}_2(S \setminus S')$, that is, if player 1 cannot force the game into S', it does not mean that player 2 can force the game into the complement of S'. For example, consider the game of *matching bits*, in which each player picks a bit in $\{0, 1\}$ simultaneously and independently. Player 1 wins if both players pick the same bit, and player 2 wins if they pick different bits. This game is not determined. For every strategy ("0" or "1") of player 1, there is a spoiling strategy ("1" and "0", respectively) of player 2. Similarly, for every strategy of player 2, player 1 has a spoiling strategy that matches the bit chosen by player 2. We shall return to this point in Sect. 31.5.

For turn-based game structures, though, determinacy holds [61].

Theorem 5 *For any turn-based game $\mathcal{G}$ and for any ω-regular property Φ, we have*

$$\langle 1\rangle^{\mathbb{B}}_{\mathcal{G}}\Phi = S \setminus \langle 2\rangle^{\mathbb{B}}_{\mathcal{G}}\big(S^{\omega} \setminus \Phi\big)$$

In fact, the theorem holds for any Borel-definable linear property Φ [76]. However, assuming the axiom of choice, one can construct games that are not determined [69].

31.4.4 From Verification to Synthesis

For player 1 or player 2 game structures, the predicates $\langle i\rangle^{\mathbb{B}}_{\mathcal{G}}\Phi$ reduce to the usual *verification* problems. For a player 1 game structure $\mathcal{G}$, we write $\exists^{\mathbb{B}}_{\mathcal{G}}\Phi$ for the set of states from which player 1 has a strategy to ensure that the outcome is in Φ. This is the usual (existential) verification question: does there exist a trace starting from s satisfying Φ? Dually, we write $\forall^{\mathbb{B}}_{\mathcal{G}}\Phi$ for the set of states from which every player 1 strategy enforces that the outcome is in Φ. This is the usual (universal) verification question: does every trace starting from s satisfy Φ? The relation (15) gives a symbolic algorithm to solve both the existential and the universal verification problems for parity objectives. Since any linear-time ω-regular property (such as those expressed in LTL) can be reduced to a parity objective, this gives an algorithm for the verification of such properties. However, for verification problems, the reduction to parity automata is not necessary. Instead, for each ω-regular property, one can construct a nondeterministic Büchi automaton, and symbolically evaluate the Büchi property $\Box\Diamond T$ on the product of a game with the automaton [44, 54]. For a player 1 game structure $\mathcal{G}$, define

$$\mathrm{Epre}(X) = \big\{s \in S \;\big|\; \exists a \in \Gamma_1(s).\delta(s,a,\cdot) \in X\big\}$$
$$\mathrm{Apre}(X) = \big\{s \in S \;\big|\; \forall a \in \Gamma_1(s).\delta(s,a,\cdot) \in X\big\}.$$

(In the expression, we have omitted the trivial move of player 2 in the transition function.) Using these operators, we can write μ-calculus expressions for the existential and universal verification problems. For example,

$$\exists^{\mathbb{B}}_{\mathcal{G}}\Diamond T = \big[\!\big[\mu x.T \vee \mathrm{Epre}(x)\big]\!\big] \quad \text{and} \quad \forall^{\mathbb{B}}_{\mathcal{G}}\Diamond T = \big[\!\big[\mu x.T \vee \mathrm{Apre}(x)\big]\!\big].$$

Clearly, a generic way to get verification algorithms is to convert to parity objectives and then use (15), noting that Epre and Apre are special cases for games where only player 1 or player 2 has nontrivial choice of moves.

Conversely, can one take a μ-calculus formula with Epre or Apre, systematically replace each Epre (or Apre) by pre_1, and solve the corresponding game problem? In general, this is not possible. Consider the co-Büchi property $\Diamond\Box T$. The following is known:

$$\begin{aligned}\exists^{\mathbb{B}}\Diamond\Box T &= \mu x.\big(\mathrm{Epre}(x) \vee \big(\nu y.T \wedge \mathrm{Epre}(y)\big)\big)\\ &= \mu x.\big(\mathrm{Epre}(x) \vee \big(T \wedge \mathrm{Epre}\big(\nu y.\big(T \wedge \mathrm{Epre}(y)\big)\big)\big)\big).\end{aligned}$$

The first formula can be obtained following the construction in [54], the second from the translation in [18, 44]. However, consider the following game $\mathcal{G}$. The set of states is $\{s_0, s_1, s_2\}$ and the target $T = \{s_0, s_2\}$. At s_0, player 2 has a choice of two moves and player 1 has no choice: player 2 can either keep the game in s_0 or move to s_1. At s_1, player 1 has a choice of two moves and player 2 has no choice: player 1 can either stay in s_1 or move to s_2. State s_2 is a sink state: once reached, the game stays there forever no matter what the players choose to do. It is easy to see that $\langle 1\rangle^{\mathbb{B}}_{\mathcal{G}} \Diamond\Box T = \{s_0, s_1, s_2\}$. Intuitively, either player 2 keeps the game forever in s_0 (in which case $\Box T$ holds, and hence $\Diamond\Box T$ holds), or at some point moves to s_1. However, once the game is in s_1, player 1 can force a move to s_2 and again ensures that eventually $\Diamond\Box T$ holds. However, a straightforward calculation shows that both the μ-calculus formulas above compute the set $\{s_1, s_2\}$ when Epre is replaced with pre_1.

A Boolean μ-calculus formula is called a *player 1 formula* if it only uses the pre_1 operator from $\mathcal{F}$. The relation between verification and control algorithms is given by the *extremal model* theorem [7], which states that a player 1 Boolean μ-calculus solves the game with an ω-regular objective Φ if and only if the formula obtained by replacing all occurrences of pre_1 with Epre solves the existential verification problem, and the formula obtained by replacing all occurrences of pre_1 with Apre solves the universal verification problem. The two verification problems are "extremal", or one-sided, versions of games in which one or the other player has no choices. The theorem states that a formula that works for all extremal cases of games also works for all games.

Theorem 6 ([7]) *For all linear ω-regular properties Φ and player* 1 *Boolean μ-calculus formulas φ, the following are equivalent*:

- $\langle 1\rangle^{\mathbb{B}}_{\mathcal{G}} \Phi = [\![\varphi(\mathrm{pre}_1)]\!]_{\mathcal{G}}$
- $\exists^{\mathbb{B}}_{\mathcal{G}} \Phi = [\![\varphi(\mathrm{Epre})]\!]_{\mathcal{G}}$ *and* $\forall^{\mathbb{B}}_{\mathcal{G}} \Phi = [\![\varphi(\mathrm{Apre})]\!]_{\mathcal{G}}$.

For the co-Büchi property $\Diamond\Box T$, we have that

$$\langle 1\rangle^{\mathbb{B}}_{\mathcal{G}} \Diamond\Box T = \mu x.\nu y.\big(\mathrm{pre}_1(x) \vee \big(T \wedge \mathrm{pre}_1(y)\big)\big)$$

and as expected,

$$\exists^{\mathbb{B}}_{\mathcal{G}} \Diamond\Box T = \mu x.\nu y.\big(\mathrm{Epre}(x) \vee \big(T \wedge \mathrm{Epre}(y)\big)\big)$$

$$\forall^{\mathbb{B}}_{\mathcal{G}} \Diamond\Box T = \mu x.\nu y.\big(\mathrm{Apre}(x) \vee \big(T \wedge \mathrm{Apre}(y)\big)\big).$$

31.4.5 Examples of Synthesis

For each class of examples considered in Sect. 31.3, one can study the corresponding symbolic synthesis question. We briefly outline a few instances.

Program Synthesis. In program synthesis, the user provides a specification and a partial implementation, and the synthesis engine provides a full implementation satisfying the specification. We can generalize the guarded command language from Sect. 31.3 to additionally include a set C of *control variables* whose values are determined as a function of the input variables I and the program variables X. Now, the semantics of a program is given by a game structure where one player determines the program inputs I and the other player determines the control variables C. Logical constraints still provide a region algebra, but notice that the formulas may now have quantifier alternations.

In recent years, there have been several novel applications of program synthesis. For example, in the synthesis of digital filters [85, 86], the programmer provides a skeleton implementation with some "holes" and asks whether there is a way to fill in the holes with constants or expressions so that the input-output behavior satisfies a given specification. The corresponding synthesis question reduces to finding satisfying assignments to an exists-forall formula. This idea, commonly called *program sketching*, has seen many interesting applications and continues to be an active area of research [21, 60, 87]. A second and related application of synthesis is in *program repair*, where the user synthesizes additional code that, when combined with a given program, ensures that the combination satisfies a given specification [36, 68, 70].

Antichain Algorithms for Synthesis. Antichain algorithms have been extended to alternating automata [51, 52]. The resulting symbolic algorithms provide the basis for symbolic synthesis procedures for LTL objectives [56, 57], where they often outperform synthesis procedures based on binary decision diagrams. The Acacia tool implements synthesis procedures based on region algebras over antichains [22].

Timed and Hybrid Games. Timed and hybrid automata were generalized to timed and hybrid games [13, 75, 94] as models for controller synthesis for timed and hybrid systems. Synthesis for linear objectives remains decidable for timed games, and region-based strategies (i.e., in which the player plays the same action from all states in a region) are sufficient for timed games. Symbolic algorithms for verification generalize to games. For hybrid systems in general, the symbolic algorithms need not converge (the verification problem is already undecidable), but the algorithms do converge for initialized rectangular games [63]. Symbolic algorithms can explicitly account for Zeno strategies, which enable a player to win by preventing the progression of time [3]. Efficient implementations of symbolic synthesis algorithms for timed games have been developed, e.g., in the Uppaal-Tiga tool [17, 35], and applied to significant industrial case studies.

31.5 Probabilistic Systems

We give a final example of symbolic techniques for non-Boolean domains: probabilistic games. In a probabilistic game, the transition relation is probabilistic, and, in

addition, players are allowed to play randomized strategies. A randomized strategy prescribes to every history of a game a probability distribution over moves (rather than a single move).

Even for games where the transition relation is deterministic, adding randomization to strategies can be helpful. Consider again the *matching bits* game from Sect. 31.4. Suppose we add randomization to strategies, that is, allow the players to choose a bit in $\{0, 1\}$ by sampling from a probability distribution. If player 1 picks a bit uniformly at random, then with probability $\frac{1}{2}$ the bits match (no matter what strategy player 2 chooses). If the game of matching bits is played repeatedly, then the probability that player 1 matches bits in some round is 1, that is, player 1 can almost surely ensure that eventually the bits match.

31.5.1 Probabilistic Games and Objectives

For a finite set A, a *probability distribution* on A is a function $p : A \to [0, 1]$ such that $\sum_{a \in A} p(a) = 1$. We denote the set of probability distributions on A by $\mathcal{D}(A)$.

A (two-player) *probabilistic game structure* $\mathcal{G} = (S, M, \Gamma_1, \Gamma_2, \delta)$ consists of a *finite* set S of states, a finite set M of moves, and move assignments $\Gamma_1, \Gamma_2 : S \to 2^M \setminus \emptyset$ as for deterministic games, and a probabilistic transition function δ that gives the probability $\delta(t \mid s, m_1, m_2)$ of a transition from s to t when moves m_1 and m_2 are chosen, for all $s, t \in S$ and all moves $m_1 \in \Gamma_1(s)$ and $m_2 \in \Gamma_2(s)$.

At every state $s \in S$, player 1 chooses a move $m_1 \in \Gamma_1(s)$, and simultaneously and independently player 2 chooses a move $m_2 \in \Gamma_2(s)$. The game then proceeds to the successor state t with probability $\delta(t \mid s, m_1, m_2)$, for all $t \in S$. As with deterministic games, we assume that the players act *non-cooperatively*, i.e., each player chooses her strategy independently and secretly from the other player, and is only interested in maximizing her own reward.

Note that deterministic game structures are a special case, where for each s, m_1, and m_2, there is some $t \in S$ such that $\delta(t \mid s, m_1, m_2) = 1$. A probabilistic game structure is *turn-based* if at any state at most one player has a non-trivial choice of moves, that is, if for each $s \in S$ either $|\Gamma_1(s)| = 1$ or $|\Gamma_2(s)| = 1$.

A *strategy* for player $i \in \{1, 2\}$ is a mapping $\xi_i : S^+ \mapsto \mathcal{D}(M)$ that associates with every nonempty finite sequence $\pi \in S^+$ of states, representing the past history of the game, a probability distribution $\xi_1(\pi)$ used to select the next move. Thus, the choice of the next move can be history-dependent and randomized. The strategy ξ_i can prescribe only moves that are available to player i; that is, for all sequences $\pi \in S^*$ and states $s \in S$, we require that $\xi_i(\pi s)(m) > 0$ only if $m \in \Gamma_i(s)$. We denote by Ξ_i the set of all strategies for player $i \in \{1, 2\}$.

We get deterministic strategies as a special case: a strategy ξ is deterministic if for all $\pi \in S^+$ there is an $m \in M$ such that $\xi(\pi)(m) = 1$.

Once the starting state s and the strategies ξ_1 and ξ_2 for the two players have been chosen, the game is reduced to an ordinary stochastic process. Hence, the probabilities of events are uniquely defined, where an *event* is a measurable set of paths

sharing the same initial state. For an event $\mathcal{A}$ whose paths all start at s, we denote by $\Pr_s^{\xi_1,\xi_2}(\mathcal{A})$ the probability that a path belongs to $\mathcal{A}$ when the game starts from s and the players use the strategies ξ_1 and ξ_2. Similarly, for a measurable function f that associates a number in $\mathbb{R} \cup \{\infty\}$ with each path, we denote by $\mathrm{E}_s^{\xi_1,\xi_2}\{f\}$ the expected value of f when the game starts from s and the strategies ξ_1 and ξ_2 are used.

For example, let Φ be a linear ω-regular property. By abuse of notation we omit the starting state when writing an event, and also denote by Φ the set of paths $\pi \in \Omega$ that satisfy Φ; this set is measurable for any choice of strategies for the two players [91]. Hence, the probability that a path satisfies Φ starting from state $s \in S$ under strategies ξ_1, ξ_2 for the two players is $\Pr_s^{\xi_1,\xi_2}(\Phi)$.

We denote by Θ_i the random variable representing the i-th state of a path, that is, Θ_i is a variable that assumes value s_i on the path $s_0 s_1 s_2 \ldots$.

We again consider winning objectives given by linear ω-regular properties, and aim to calculate the maximal probability with which player $i \in \{1, 2\}$ can ensure that the property Φ holds from a state s. We call this probability the *value of the game Φ at s for player $i \in \{1, 2\}$*. This value for player 1 is given by the function $\langle 1 \rangle \Phi : S \mapsto [0, 1]$, defined for all $s \in S$ by

$$\langle 1 \rangle \Phi(s) = \sup_{\xi_1 \in \Xi_1} \inf_{\xi_2 \in \Xi_2} \Pr_s^{\xi_1,\xi_2}(\Phi).$$

Winning for player 2 is defined analogously by exchanging the roles of player 1 and player 2.

Concurrent games with ω-regular winning conditions satisfy a *quantitative* version of determinacy [77], stating that for all linear ω-regular properties Φ and all $s \in S$, we have

$$\langle 1 \rangle \Phi(s) = 1 - \langle 2 \rangle \neg \Phi(s).$$

In fact, this result holds for any Borel property Φ as well [77].

31.5.2 The Quantitative μ-calculus

Let $\mathbb{L}_{\mathbb{R}} = ([0, 1]^S, \leq)$ be the lattice of functions from S to the real interval $[0, 1]$, ordered by pointwise ordering, i.e., $f \leq g$ if for all $s \in S$ we have $f(s) \leq g(s)$. The join and merge operations are defined in the natural way:

$$(f \vee g)(s) = \max\{f(s), g(s)\}$$
$$(f \wedge g)(s) = \min\{f(s), g(s)\}$$

for all $s \in S$. We denote by $\mathbf{0}$ and $\mathbf{1}$ the constant functions that map all states into 0 and 1, respectively. We define the negation operation $\sim$ as $(\sim f)(s) = \mathbf{1} - f(s)$ for all $s \in S$.

Given a subset $Q \subseteq S$ of states, by abuse of notation we denote also by Q the *characteristic function* of Q, defined by $Q(s) = 1$ if $s \in Q$ and $Q(s) = 0$ otherwise.

Let $\mathcal{G}$ be a probabilistic game structure and let $\mathcal{P}$ be a set of propositions ranging over subsets of states of $\mathcal{G}$. We define a quantitative μ-calculus with two operators, Ppre_1 and Ppre_2, which are duals, and give its semantics over the lattice $\mathbb{L}_{\mathbb{R}}$.

The *quantitative predecessor operators* $\mathrm{Ppre}_1, \mathrm{Ppre}_2 : \mathcal{F} \mapsto \mathcal{F}$ are defined for every $f \in \mathcal{F}$ by

$$\mathrm{Ppre}_i(f)(s) = \sup_{\xi_i \in \Xi_i} \inf_{\xi_{3-i} \in \Xi_{3-i}} \mathrm{E}_s^{\xi_i, \xi_{3-i}} \{ f(\Theta_1) \}.$$

Intuitively, the value $\mathrm{Ppre}_i(f)(s)$ is the maximum expectation for the next value of f that player $i \in \{1, 2\}$ can achieve. Given $f \in \mathcal{F}$ and $i \in \{1, 2\}$, the function $\mathrm{Ppre}_i(f)$ can be computed by solving the following *matrix game* at each $s \in S$:

$$\mathrm{Ppre}_1(f)(s) = v_1 \left[\sum_{t \in S} f(t)\delta(t \mid s, a_1, a_2) \right]_{a_1 \in \Gamma_1(s), a_2 \in \Gamma_2(s)},$$

where $v_1 A$ denotes the value obtained by player 1 in the matrix game A. The existence of solutions to the above matrix games, and the existence of optimal randomized strategies for players 1 and 2, is guaranteed by the minmax theorem [81]. The matrix games may be solved using traditional linear programming algorithms (see, e.g., [82]). From properties of matrix games, we get that

$$\mathrm{Ppre}_1(f) = \mathbf{1} - \mathrm{Ppre}_2(\mathbf{1} - f)$$

for all $f : S \to [0, 1]$. That is, the operators Ppre_1 and Ppre_2 are dual.

The following result generalizes Theorem 4 to probabilistic games. Intuitively, it states that the value of a probabilistic parity game can be characterized as a nested fixpoint formula evaluated over $\mathbb{L}_{\mathbb{R}}$.

Theorem 7 ([9]) *For every probabilistic game structure $\mathcal{G}$, partition $T_1, \ldots, T_k$ of the states of $\mathcal{G}$, and players $i \in \{1, 2\}$, we have*

$$\langle 1 \rangle_{\mathcal{G}}^{\mathbb{B}} \mathrm{Parity}(T_1, \ldots, T_k) = \left[\!\left[\lambda_k x_k \ldots \nu x_2 \mu x_1 . \bigvee_{j=1}^{k} \left(T_i \wedge \mathrm{pre}_i(x_j) \right) \right]\!\right] \tag{16}$$

where $\lambda_k = \nu$ if k is even and $\lambda_k = \mu$ if k is odd.

Note that even though the game structure has finitely many states, the above characterization does not give a terminating algorithm for parity games. Instead, the best known algorithms go through a careful analysis of formulas in the theory of reals [9, 73]. A better NP ∩ co-NP complexity bound is known for the special case of *turn-based games* in which players alternate in selecting moves, that is, when $\Gamma_1(s)$ or $\Gamma_2(s)$ is a singleton for each state s [37, 41]. Curiously, the best known

complexity lower bound of this very general class of games is the same as that of alternating reachability (P-hard).

While we focussed on a quantitative interpretation, probabilistic games can also be studied relative to a *qualitative* interpretation. Intuitively, qualitative winning objectives state that a player wins on all or almost all (in the sense of measure theory) paths, while quantitative winning objectives measure the exact probability of winning. Qualitative notions of winning also admit fixpoint characterizations using an appropriately defined μ-calculus, with somewhat complex pre operations [5, 6, 38, 41].

31.6 Conclusion

In this chapter we have taken a general view of symbolic model checking as computing fixpoints over lattices, which gives a unified approach applicable to many different classes of systems and properties. The basis for this unification is the notion of region algebras together with appropriate generalizations of the μ-calculus. Region algebras provide a nice abstraction that allows us to separate the semantics of a transition structure from the data structure that is used to algorithmically manipulate this transition structure. The μ-calculus is a powerful formalism for specifying properties of sets of states of a transition system that are fixpoints of monotone functions. We have applied this general framework to a variety of problems both in verification and synthesis.

First, we have considered the verification of properties of labelled transition systems that are not necessarily finite and shown how classical linear-time properties can be expressed in the μ-calculus. When a data structure provides effectiveness properties to the region algebra, iterated approximations provide natural semi-algorithms to evaluate μ-calculus expressions. While transfinite number of approximations are in general necessary to obtain the exact value of expressions, we have provided conditions, in the form of equivalence relations on the underlying state space of the transition system, that ensure that a finite number of approximations is sufficient. We have shown that this framework is rich enough to explain decidability results and symbolic verification techniques for infinite-state systems such as timed automata, hybrid rectangular automata, or Petri nets. We have also shown that this framework can be applied to obtain new efficient algorithms for solving problems whose underlying transition system is finite: as an illustration, we have shown how to solve the universality problem for nondeterministic finite automata using fixpoint computations paired with antichains of sets of states that are used to represent the underlying state space. For this problem, this data structure performs better than binary decision diagrams. This shows that even in the finite state case, it is sometimes useful to look for alternatives to BDDs for performing symbolic verification.

Second, we have shown that the region algebra and μ-calculus framework can be applied to several extensions to the notion of transition systems. We have considered deterministic game structures that can be used to formalize synthesis problems as

two-player zero-sum ω-regular games and probabilistic game structures that generalize classical models like Markov chains and Markov decision processes. To cover these rich classes of transition systems, we have shown how to extend the μ-calculus in an appropriate way.

In this chapter, it was not possible to cover all the research directions that are explored in the area of symbolic verification techniques. We list a few other directions where symbolic techniques have played, or continue to play, a central role:

- Regular model checking. While, as we have shown in this chapter, linear constraints are well-studied data structures for representing numerical state spaces, other techniques have been proposed: finite automata have also been explored to manipulate Presburger definable sets [93], see also [24] for richer numerical domains. More generally, automata have been used as a data structure for the algorithmic manipulation of infinite-state systems in the framework of regular model checking [26].
- Acceleration techniques. As we have seen, the termination of fixpoint computations that use successive approximations can only be ensured for subclasses of infinite-state systems. To obtain termination in practice, even for classes of systems where termination cannot be ensured in all cases, acceleration techniques have been proposed. These techniques try, for example, to compute the repeated application of linear transformations [15, 23] or the repeated application of transducers in the context of regular model checking [27].
- Approximation techniques. In the vocabulary of abstract interpretation [42], region algebras represent the collecting semantics of the transition systems that they represent. Abstract interpretation provides a general framework for abstracting such semantics into simpler ones, called abstract semantics, in order to define approximate analysis algorithms. Abstract interpretation is also concerned with the definition of data structures for efficient representation of abstract domains, such as octagons [80] for the approximation of more general numerical domains, and with acceleration techniques, known there as widenings [43], for extrapolating the limits of fixpoint computation.
- Quantitative verification. A recent direction generalizes logics for specification to *quantitative* logics, whose semantics gives a numerical value to a property, rather than a Boolean [2, 4, 19, 39, 55]. While a theory of quantitative specifications is still under active development and refinement, symbolic techniques have played a central role in the analysis.
- Equivalences and metrics. While we focused on symbolic techniques for verification and synthesis, they also play a role in defining and computing behavioral equivalences and their generalizations to metrics on systems. For example, classical behavioral equivalences such as bisimilarity and similarity [79] and their game analogues [12] are captured by fixpoint formulas. More recently, similar fixpoint characterizations have been given for metrics on (stochastic) systems [10, 30, 47, 48].

Finally, we point out other surveys that present results related to symbolic model checking for the Boolean case and beyond [2, 40]. These surveys contain several

results that are complementary to those presented here as well as a list of relevant pointers to the literature.

Acknowledgements We would like to thank Emmanuel Filiot and Guillermo A. Pérez for carefully reading a previous version of this chapter.

References

1. Abdulla, P.A., Čerāns, K., Jonsson, B., Tsay, Y.K.: General decidability theorems for infinite-state systems. In: Symp. on Logic in Computer Science (LICS), pp. 313–321. IEEE, Piscataway (1996)
2. de Alfaro, L.: Quantitative verification and control via the mu-calculus. In: Amadio, R., Lugiez, D. (eds.) Intl. Conf. on Concurrency Theory (CONCUR). LNCS, vol. 2761, pp. 102–126. Springer, Heidelberg (2003)
3. de Alfaro, L., Faella, M., Henzinger, T., Majumdar, R., Stoelinga, M.: The element of surprise in timed games. In: Amadio, R., Lugiez, D. (eds.) Intl. Conf. on Concurrency Theory (CONCUR). LNCS, vol. 2761, pp. 142–156. Springer, Heidelberg (2003)
4. de Alfaro, L., Faella, M., Henzinger, T., Majumdar, R., Stoelinga, M.: Model checking discounted temporal properties. Theor. Comput. Sci. **345**(1), 139–170 (2005)
5. de Alfaro, L., Henzinger, T.: Concurrent omega-regular games. In: Symp. on Logic in Computer Science (LICS), pp. 141–154. IEEE, Piscataway (2000)
6. de Alfaro, L., Henzinger, T., Kupferman, O.: Concurrent reachability games. In: Annual Symp. on Foundations of Computer Science (FOCS), pp. 564–575. IEEE, Piscataway (1998)
7. de Alfaro, L., Henzinger, T., Majumdar, R.: From verification to control: dynamic programs for omega-regular objectives. In: Symp. on Logic in Computer Science (LICS), pp. 279–290. IEEE, Piscataway (2001)
8. de Alfaro, L., Henzinger, T.A., Majumdar, R.: Symbolic algorithms for infinite-state games. In: Larsen, K.G., Nielsen, M. (eds.) Intl. Conf. on Concurrency Theory (CONCUR). LNCS, vol. 2154, pp. 536–550 (2001)
9. de Alfaro, L., Majumdar, R.: Quantitative solution of omega-regular games. J. Comput. Syst. Sci. **68**(2), 374–397 (2004)
10. de Alfaro, L., Majumdar, R., Raman, V., Stoelinga, M.: Game refinement relations and metrics. Log. Methods Comput. Sci. **4**(3) (2008)
11. Alur, R., Dill, D.: A theory of timed automata. Theor. Comput. Sci. **126**, 183–235 (1994)
12. Alur, R., Henzinger, T., Kupferman, O., Vardi, M.: Alternating refinement relations. In: Sangiorgi, D., de Simone, R. (eds.) Intl. Conf. on Concurrency Theory (CONCUR). LNCS, vol. 1466, pp. 163–178. Springer, Heidelberg (1998)
13. Asarin, E., Maler, O., Pnueli, A.: Symbolic controller synthesis for discrete and timed systems. In: Antsaklis, P., Nerode, A., Kohn, W., Sastry, S. (eds.) Hybrid Systems (II). LNCS, vol. 999, pp. 1–20. Springer, Heidelberg (1995)
14. Bagnara, R., Hill, P.M., Zaffanella, E.: The Parma Polyhedra Library: toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Sci. Comput. Program. **72**(1–2), 3–21 (2008)
15. Bardin, S., Finkel, A., Leroux, J., Petrucci, L.: FAST: acceleration from theory to practice. Int. J. Softw. Tools Technol. Transf. **10**(5), 401–424 (2008)
16. Basu, S.: New results on quantifier elimination over real closed fields and applications to constraint databases. J. ACM **46**(4), 537–555 (1999)
17. Behrmann, G., Cougnard, A., David, A., Fleury, E., Larsen, K., Lime, D.: UPPAAL-Tiga: time for playing games! In: Damm, W., Hermanns, H. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 4590, pp. 121–125. Springer, Heidelberg (2007)

18. Bhat, G., Cleaveland, R.: Efficient local model-checking for fragments of the modal μ-calculus. In: Margaria, T., Steffen, B. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 1055, pp. 107–126. Springer, Heidelberg (1996)
19. Bloem, R., Chatterjee, K., Henzinger, T., Jobstmann, B.: Better quality in synthesis through quantitative objectives. In: Bouajjani, A., Maler, O. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 5643, pp. 140–156. Springer, Heidelberg (2009)
20. Bloem, R., Jobstmann, B., Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of reactive(1) designs. J. Comput. Syst. Sci. **78**(3), 911–938 (2012)
21. Bodík, R., Chandra, S., Galenson, J., Kimelman, D., Tung, N., Barman, S., Rodarmor, C.: Programming with angelic nondeterminism. In: Hermenegildo, M.V., Palsberg, J. (eds.) Symp. on Principles of Programming Languages (POPL), pp. 339–352. ACM, New York (2010)
22. Bohy, A., Bruyère, V., Filiot, E., Jin, N., Raskin, J.F.: Acacia+, a tool for LTL synthesis. In: Madhusudan, P., Seshia, S. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 7358, pp. 652–657. Springer, Heidelberg (2012)
23. Boigelot, B.: Domain-specific regular acceleration. Int. J. Softw. Tools Technol. Transf. **14**(2), 193–206 (2012)
24. Boigelot, B., Jodogne, S., Wolper, P.: On the use of weak automata for deciding linear arithmetic with integer and real variables. In: Goré, R., Leitsch, A., Nipkow, T. (eds.) Intl. Joint Conf. on Automated Reasoning (IJCAR). LNCS, vol. 2083. Springer, Heidelberg (2001)
25. Bouajjani, A., Habermehl, P., Holík, L., Touili, T., Vojnar, T.: Antichain-based universality and inclusion testing over nondeterministic finite tree automata. In: Ibarra, O.H., Ravikumar, B. (eds.) Implementation and Applications of Automata (CIAA). LNCS, vol. 5148, pp. 57–67. Springer, Heidelberg (2008)
26. Bouajjani, A., Jonsson, B., Nilsson, M., Touili, T.: Regular model checking. In: Emerson, E.A., Sistla, A.P. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 1855. Springer, Heidelberg (2000)
27. Bouajjani, A., Touili, T.: Extrapolating tree transformations. In: Brinksma, E., Larsen, K.G. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 2404. Springer, Heidelberg (2002)
28. Bouyer, P.: Untameable timed automata! In: Alt, H., Habib, M. (eds.) Annual Symposium on Theoretical Aspects of Computer Science. LNCS, vol. 2607, pp. 620–631. Springer, Heidelberg (2003)
29. Bradfield, J.: The modal μ-calculus alternation hierarchy is strict. Theor. Comput. Sci. **195**(2), 133–153 (1998)
30. van Breugel, F., Worrel, J.: Towards quantitative verification of probabilistic systems. In: Orejas, F., Spirakis, P.G., van Leeuwen, J. (eds.) Intl. Colloquium on Automata, Languages and Programming. LNCS, vol. 2076, pp. 421–432. Springer, Heidelberg (2001)
31. Bryant, R.: Graph-based algorithms for Boolean function manipulation. Trans. Comput. **C-35**(8), 677–691 (1986)
32. Bultan, T., Gerber, R., Pugh, W.: Symbolic model checking of infinite systems using Presburger arithmetic. In: Grumberg, O. (ed.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 1254, pp. 400–411. Springer, Heidelberg (1997)
33. Burch, J., Clarke, E., McMillan, K., Dill, D., Hwang, L.: Symbolic model checking: 10^{20} states and beyond. Inf. Comput. **98**(2), 142–170 (1992)
34. Bustan, D., Kupferman, O., Vardi, M.: A measured collapse of the modal μ-calculus. In: Diekert, V., Habib, M. (eds.) Annual Symposium on Theoretical Aspects of Computer Science. LNCS, vol. 2996, pp. 522–533. Springer, Heidelberg (2004)
35. Cassez, F., David, A., Fleury, E., Larsen, K., Lime, D.: Efficient on-the-fly algorithms for the analysis of timed games. In: Abadi, M., de Alfaro, L. (eds.) Intl. Conf. on Concurrency Theory (CONCUR). LNCS, vol. 3653, pp. 66–80. Springer, Heidelberg (2005)
36. Chandra, S., Torlak, E., Barman, S., Bodík, R.: Angelic debugging. In: Taylor, R.N., Gall, H.C., Medvidovic, N. (eds.) Intl. Conf. on Software Engineering (ICSE), pp. 121–130. ACM, New York (2011)

37. Chatterjee, K., de Alfaro, L., Henzinger, T.: The complexity of quantitative concurrent parity games. In: Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 678–687. ACM, New York (2006)
38. Chatterjee, K., de Alfaro, L., Henzinger, T.: Qualitative concurrent parity games. Trans. Comput. Log. **12**(4), 28 (2011)
39. Chatterjee, K., Doyen, L., Henzinger, T.: Quantitative languages. Trans. Comput. Log. **11**(4), 23:1–23:38 (2010)
40. Chatterjee, K., Henzinger, T.: Value iteration. In: Grumberg, O., Veith, H. (eds.) 25 Years of Model Checking. LNCS, vol. 5000, pp. 107–138. Springer, Heidelberg (2008)
41. Chatterjee, K., Henzinger, T.: A survey of stochastic ω-regular games. J. Comput. Syst. Sci. **78**(2), 394–413 (2012)
42. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Graham, R.M., Harrison, M.A., Sethi, R. (eds.) Symp. on Principles of Programming Languages (POPL). ACM, New York (1977)
43. Cousot, P., Cousot, R.: Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In: Bruynooghe, M., Wirsing, M. (eds.) Programming Language Implementation and Logic Programming (PLILP). LNCS, vol. 631. Springer, Heidelberg (1992)
44. Dam, M.: CTL* and ECTL* as fragments of the modal μ-calculus. Theor. Comput. Sci. **126**, 77–96 (1994)
45. De Wulf, M., Doyen, L., Henzinger, T.A., Raskin, J.F.: Antichains: a new algorithm for checking universality of finite automata. In: Ball, T., Jones, R.B. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 4144, pp. 17–30. Springer, Heidelberg (2006)
46. De Wulf, M., Doyen, L., Maquet, N., Raskin, J.F.: Antichains: alternative algorithms for LTL satisfiability and model-checking. In: Ramakrishnan, C.R., Rehof, J. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 4963, pp. 63–77. Springer, Heidelberg (2008)
47. Desharnais, J., Gupta, V., Jagadeesan, R., Panangaden, P.: Metrics for labelled Markov systems. In: Baeten, J.C.M., Mauw, S. (eds.) Intl. Conf. on Concurrency Theory (CONCUR). LNCS, vol. 1664, pp. 258–273. Springer, Heidelberg (1999)
48. Desharnais, J., Gupta, V., Jagadeesan, R., Panangaden, P.: The metric analogue of weak bisimulation for probabilistic processes. In: Symp. on Logic in Computer Science (LICS), pp. 413–422. IEEE, Piscataway (2002)
49. Dickson, L.: Finiteness of the odd perfect and primitive abundant numbers with n distinct prime factors. Am. J. Math. **35**, 413–422 (1913)
50. Dijkstra, E.: A Discipline of Programming. Prentice Hall, Englewood Cliffs (1976)
51. Doyen, L., Raskin, J.F.: Antichains for the automata-based approach to model-checking. Log. Methods Comput. Sci. **1**, 5 (2009)
52. Doyen, L., Raskin, J.F.: Antichain algorithms for finite automata. In: Esparza, J., Majumdar, R. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 6015, pp. 2–22. Springer, Heidelberg (2010)
53. Emerson, E., Jutla, C.: Tree automata, mu-calculus and determinacy. In: Annual Symp. on Foundations of Computer Science (FOCS), pp. 368–377. IEEE, Piscataway (1991)
54. Emerson, E., Lei, C.: Efficient model checking in fragments of the propositional μ-calculus. In: Symp. on Logic in Computer Science (LICS), pp. 267–278. IEEE, Piscataway (1986)
55. Faella, M., Legay, A., Stoelinga, M.: Model checking quantitative linear time logic. Electron. Notes Theor. Comput. Sci. **220**(3), 61–77 (2008)
56. Filiot, E., Jin, N., Raskin, J.F.: An antichain algorithm for LTL realizability. In: Bouajjani, A., Maler, O. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 5643, pp. 263–277. Springer, Heidelberg (2009)
57. Filiot, E., Jin, N., Raskin, J.F.: Antichains and compositional algorithms for LTL synthesis. Form. Methods Syst. Des. **39**(3), 261–296 (2011)
58. Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere. Tech. Rep. LSV-98-4, Laboratoire Spécification et Vérification (1998)

59. Frehse, G.: PHAVer: algorithmic verification of hybrid systems past HyTech. Int. J. Softw. Tools Technol. Transf. **10**(3), 263–279 (2008)
60. Gulwani, S., Jha, S., Tiwari, A., Venkatesan, R.: Synthesis of loop-free programs. In: Hall, M.W., Padua, D.A. (eds.) Conf. on Programming Language Design and Implementation (PLDI), pp. 62–73. ACM, New York (2011)
61. Gurevich, Y., Harrington, L.: Trees, automata, and games. In: Lewis, H.R., Simons, B.B., Burkhard, W.A., Landweber, L.H. (eds.) Annual Symp. on the Theory of Computing, pp. 60–65. ACM, New York (1982)
62. Henzinger, T., Ho, P.H., Wong-Toi, H.: HyTech: the next generation. In: Real-Time Systems Symposium (RTSS), pp. 56–65. IEEE, Piscataway (1995)
63. Henzinger, T., Horowitz, B., Majumdar, R.: Rectangular hybrid games. In: Baeten, J., Mauw, S. (eds.) Intl. Conf. on Concurrency Theory (CONCUR). LNCS, vol. 1664, pp. 320–335. Springer, Heidelberg (1999)
64. Henzinger, T., Kupferman, O., Qadeer, S.: From *pre*historic to *post*modern symbolic model checking. In: Hu, A., Vardi, M. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 1427, pp. 195–206. Springer, Heidelberg (1998)
65. Henzinger, T.A., Kopke, P.W., Puri, A., Varaiya, P.: What's decidable about hybrid automata? J. Comput. Syst. Sci. **57**(1), 94–124 (1998)
66. Henzinger, T.A., Majumdar, R., Raskin, J.F.: A classification of symbolic transition systems. Trans. Comput. Log. **6**(1), 1–32 (2005)
67. Jobstmann, B., Galler, S., Weiglhofer, M., Bloem, R.A.: A tool for property synthesis. In: Damm, W., Hermanns, H. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 4590, pp. 258–262. Springer, Heidelberg (2007)
68. Jobstmann, B., Staber, S., Griesmayer, A., Bloem, R.: Finding and fixing faults. J. Comput. Syst. Sci. **78**(2), 441–460 (2012)
69. Kechris, A.: Classical Descriptive Set Theory. Springer, Heidelberg (1994)
70. Könighofer, R., Bloem, R.: Automated error localization and correction for imperative programs. In: Bjesse, P., Slobodová, A. (eds.) Formal Methods in Computer Aided Design (FMCAD), pp. 91–100. FMCAD, Austin (2011)
71. Kozen, D.: Results on the propositional μ-calculus. Theor. Comput. Sci. **27**(3), 333–354 (1983)
72. Kress-Gazit, H., Wongpiromsarn, T., Topcu, U.: Correct, reactive robot control from abstraction and temporal logic specifications. Robot. Autom. Mag. **18**(3), 65–74 (2011)
73. Kristoffer, S., Frederiksen, S., Miltersen, P.: Approximating the value of a concurrent reachability game in the polynomial time hierarchy. In: Cai, L., Cheng, S.W., Lam, T. (eds.) Intl. Symposium on Algorithms and Computation (ISAAC). LNCS, vol. 8283, pp. 457–467. Springer, Heidelberg (2013)
74. Larsen, K.G., Pettersson, P., Yi, W.: Uppaal in a nutshell. Int. J. Softw. Tools Technol. Transf. **1**(1–2), 134–152 (1997)
75. Maler, O., Pnueli, A., Sifakis, J.: On the synthesis of discrete controllers for timed systems. In: Mayr, E.W., Puech, C. (eds.) Annual Symposium on Theoretical Aspects of Computer Science. LNCS, vol. 900, pp. 229–242. Springer, Heidelberg (1995)
76. Martin, D.: Borel determinacy. Ann. Math. **102**, 363–371 (1975)
77. Martin, D.: The determinacy of Blackwell games. J. Symb. Log. **63**(4), 1565–1581 (1998)
78. McMillan, K.: Symbolic Model Checking: An Approach to the State-Explosion Problem. Kluwer Academic, Norwell (1993)
79. Milner, R.: Communication and Concurrency. Prentice Hall, Upper Saddle River (1989)
80. Miné, A.: The octagon abstract domain. High.-Order Symb. Comput. **19**(1), 31–100 (2006)
81. von Neumann, J., Morgenstern, O.: Theory of Games and Economic Behavior. Princeton University Press, Princeton (1947)
82. Owen, G.: Game Theory. Academic Press, Cambridge (1995)
83. Pnueli, A.: The temporal logic of programs. In: Annual Symp. on Foundations of Computer Science (FOCS), pp. 46–57. IEEE, Piscataway (1977)
84. Reisig, W.: Petri Nets: An Introduction. Springer, Heidelberg (1985)

85. Solar-Lezama, A., Rabbah, R., Bodík, R., Ebcioglu, K.: Programming by sketching for bit-streaming programs. In: Sarkar, V., Hall, M.W. (eds.) Conf. on Programming Language Design and Implementation (PLDI), pp. 281–294. ACM, New York (2005)
86. Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S., Saraswat, V.: Combinatorial sketching for finite programs. In: Shen, J.P., Martonosi, M. (eds.) Intl. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 404–415. ACM, New York (2006)
87. Srivastava, S., Gulwani, S., Foster, J.: From program verification to program synthesis. In: Hermenegildo, M.V., Palsberg, J. (eds.) Symp. on Principles of Programming Languages (POPL), pp. 313–326. ACM, New York (2010)
88. Stockmeyer, L.: The complexity of decision problems in automata theory and logic. Ph.D. thesis, Massachusetts Institute of Technology (1974)
89. Thomas, W.: Automata on infinite objects. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, vol. B, pp. 133–191. Elsevier, Amsterdam (1990)
90. Thomas, W.: On the synthesis of strategies in infinite games. In: Mayr, E.W., Puech, C. (eds.) Annual Symposium on Theoretical Aspects of Computer Science. LNCS, vol. 900, pp. 1–13. Springer, Heidelberg (1995)
91. Vardi, M.: Automatic verification of probabilistic concurrent finite-state systems. In: Annual Symp. on Foundations of Computer Science (FOCS), pp. 327–338. IEEE, Piscataway (1985)
92. Vardi, M., Wolper, P.: Reasoning about infinite computations. Inf. Comput. **115**(1), 1–37 (1994)
93. Wolper, P., Boigelot, B.: An automata-theoretic approach to Presburger arithmetic constraints (extended abstract). In: Mycroft, A. (ed.) Intl. Symp. on Static Analysis (SAS). LNCS, vol. 983. Springer, Heidelberg (1995)
94. Wong-Toi, H.: The synthesis of controllers for linear hybrid automata. In: Decision and Control, pp. 4607–4612. IEEE, Piscataway (1997)

Chapter 32
Process Algebra and Model Checking

Rance Cleaveland, A.W. Roscoe, and Scott A. Smolka

Abstract Process algebras such as CCS, CSP and ACP are abstract notations for describing concurrent systems that interact via (usually) handshake-based communication. They lead to natural concepts of process state and are therefore natural candidates for model checking. We survey the area of process algebra and model checking, focusing on these three process algebras. We first introduce the syntax and semantics of these process algebras, before looking at the algorithmic basis for their model checking, which includes ideas such as bisimulation and refinement as well as the logics used to describe system-correctness properties. Finally, we introduce the process-alebra-based model-checking tools FDR, CWB and XMC, illustrating their utility by a number of case studies.

32.1 Introduction

Process algebra [10] refers to a class of algebraic formalisms for modeling and reasoning about concurrent systems of processes. The hallmarks of process algebra include a collection of operators for composing systems out of subsystems, and an equivalence or refinement relation for determining respectively when two systems exhibit the same behavior or when one system's behavior is more constrained than another's. The field draws its inspiration, and name, from the mathematical study of so-called *universal algebra*, and was first studied intensively in the late 1970s and early 1980s. The Calculus of Communicating Systems (CCS), the theoretical version of Communicating Sequential Processes (CSP), and the Algebra of Communicating Processes (ACP) were among the earliest, and most heavily studied, process algebras.

R. Cleaveland
University of Maryland, College Park, College Park, MD, USA

A.W. Roscoe (✉)
University of Oxford, Oxford, UK
e-mail: bill.roscoe@cs.ox.ac.uk

S.A. Smolka
Stony Brook University, Stony Brook, NY, USA

E.M. Clarke et al. (eds.), *Handbook of Model Checking*,
DOI 10.1007/978-3-319-10575-8_32

Unlike model-checking approaches to verification, in which systems are checked against formulae given in temporal logic, process algebra emphasizes the use of (higher-level) system descriptions as specifications for (more detailed) system specifications. Despite this apparently fundamental difference, researchers have identified aspects of synergy and similarity between the frameworks. At the mathematical level, for example, relationships between various temporal logics and behavioral equivalences and refinement orderings have been established. These so-called *logical characterization* results typically show that equivalent systems are guaranteed to satisfy exactly the same temporal formulae, and vice versa. Such results can be used to improve the performance of model checkers, since a (smaller, easier-to-analyze) equivalent system may be substituted for another (larger, more complex) one when checking the truth of a temporal formula.

In addition, the equivalence and refinement relations used in process algebras are *compositional*, meaning that related subsystems may be replaced by one another inside a larger system, with the relationship between the subsystems "carrying over" to the larger system. This also has significant practical implications for model checking.

Other work has established converse results: for some temporal logics, one may check that a system satisfies formulae in the logic by embedding the system in a larger system and checking for equivalence with an appropriate specification. Still other work has shown how composite theories combining both logical and process operators may be constructed and refinement relations defined that seamlessly combine logical notions of satisfaction with algebraic ones of behavioral elaboration. At an algorithmic level, the computational and algorithmic foundations of model checking that allow one to efficiently check whether or not a system satisfies a formula while mitigating state explosion have also found widespread application in equivalence and refinement checking. These include compositional methods such as state-space minimization, as discussed later in the present chapter, data independence [25, 50], and symmetry and partial-order reductions [52].

This chapter begins by examining the theoretical foundations of process algebra, including the seminal CCS, CSP, and ACP algebras and their corresponding behavioral equivalences and refinement orderings (Sect. 32.2). It then considers the algorithms and methodologies used to decide these relationships, including compositional verification (Sect. 32.3). Next, the process-algebra-based tool sets FDR, CWB (the Concurrency Workbench), XMC, and mCRL2 are presented (Sect. 32.4), along with applications to several noteworthy case studies (Sect. 32.5). Finally, our concluding remarks are given (Sect. 32.6).

32.2 Foundations

This section reviews the classical foundations of process algebra. Early in the development of the field, three schools of thought emerged on the subject, roughly based around the three approaches to programming-language semantics: operational, denotational, and axiomatic. That is, while all three frameworks for process algebra

emphasized system modeling via composition operators, and the use of equivalences and preorders (refinement orderings) for relating system descriptions on the basis of their observable behavior, the methods by which the operators and behavioral relations were given precise mathematical definitions differed significantly. CCS, for example, favors operational accounts of process operators, with an equivalence then defined after the fact. CSP, on the other hand, focuses on a denotational approach: a mathematical notion of *process* is defined, with an implicit ordering relation given on these mathematical objects inherited from the definition; then operators are defined as constructors over these semantic objects. ACP, finally, follows the axiomatic approach; operators are specified and equational axioms given capturing the behavioral relationships that emerge from using these operators. The legitimacy of these axioms is then demonstrated via the construction of mathematical models for which the axioms constitute a sound and complete equational proof system.

32.2.1 CCS: Process Algebra via Operational Semantics

This section introduces the syntax of CCS and its operational semantics. The latter is given in the Structural Operational Semantics (SOS) style formalized by Plotkin [67]; inference rules are given that in effect precisely determine what the initial execution steps are of a CCS term. It then defines a notion of semantic equivalence, based on this operational definition. A logical characterization of this equivalence is given using a simple modal logic, Hennessy–Milner Logic, and derived equivalences then presented that abstract from internal computation.

32.2.1.1 Syntax of CCS

CCS [58, 59] introduces a small set of operators for constructing system descriptions from definitions of subsystems. The basic building blocks of these descriptions, and indeed of system definitions in all existing process algebras, are *actions*. Intuitively, actions represent atomic, uninterruptible activities that systems may perform, with some actions denoting internal execution and others representing potential interactions with its environment that the system may engage in.

Actions in CCS

A binary, synchronous model of process communication underlies CCS, and the structure of the set of actions reflects this design decision. Actions represent either inputs/outputs on *ports* or internal computation steps. Actions on ports are sometimes called *external*, as they require interaction with the environment in order to take place.

To formalize these intuitions, let Λ represent a countably infinite set of labels, or ports, not containing the distinguished symbol τ. Then an action in CCS has one of the following three forms.

- α, where $\alpha \in \Lambda$, represents the act of receiving a signal on port α.
- $\overline{\alpha}$, where $\alpha \in \Lambda$, represents the act of emitting a signal on port α.
- τ represents an internal computation step.

In what follows, we use A_{CCS} to stand for the set of all CCS actions; that is,

$$A_{CCS} = \Lambda \cup \{\overline{\alpha} \mid \alpha \in \Lambda\} \cup \{\tau\}.$$

We also abuse notation by defining $\overline{\overline{\alpha}} = \alpha$; note that $\overline{\tau}$ is not a valid action. We refer to the actions α and $\overline{\alpha}$, where $\alpha \in \Lambda$, as *complementary*, as they represent an input and output action on the same channel. The set $A_{CCS} - \{\tau\}$ then contains the set of external, or visible, actions; the only internal action is τ.

CCS Operators

Having defined the set A_{CCS} of CCS actions we now introduce the operators the process algebra provides for building systems. In what follows, we assume that p, p_1 and p_2 denote CCS system descriptions that have previously been constructed, and we also assume a countably infinite set $\mathscr{C}$ of *process variables*. CCS provides the following constructors.

- *nil* represent the terminated process that has finished execution.
- Given $a \in A_{CCS}$, the *prefixing* operator $a.$ allows an action to be "prepended" onto an existing system description. Intuitively, $a.p$ is capable first of an a and then behaves like p.
- $+$ represents a *choice* construct. The system $p_1 + p_2$ offers the potential of behaving like either p_1 or p_2, depending on the interactions enabled by the environment.
- $|$ denotes *parallel composition*. The system $p_1|p_2$ interleaves the execution of p_1 and p_2 while also permitting complementary actions of p_1 and p_2 to synchronize; in this case, the resulting composite action is a τ.
- If $L \subseteq A_{CCS} - \{\tau\}$ then the *restriction* operator $\backslash L$ permits actions to be localized within a system. Intuitively, $p\backslash L$ behaves like p except that it is disallowed from interacting with its environment using actions mentioned in L. Note that τ can never be restricted.
- The operator $[f]$ allows actions in a process to be *renamed*. Here f is a function from A_{CCS} to A_{CCS} that is required to satisfy the following two restrictions.
  - $f(\tau) = \tau$
  - $f(\overline{a}) = \overline{f(a)}$.

  When this is the case, f is called a *renaming*. The system $p[f]$ behaves exactly like p except that f is applied to each action that p may engage in.
- If $C \in \mathscr{C}$, then C represents a valid system provided that a *defining equation* of the form $C \stackrel{\Delta}{=} p$ has been given. Intuitively, C represents an "invocation" that behaves like p. This construct allows systems to be defined recursively.

In process-algebraic parlance, system descriptions built using the above operators are often referred to as *terms* or *processes*. We use $\mathscr{P}_{CCS}$ to represent the set of all CCS processes.

32.2.1.2 The Operational Semantics of CCS

In the account so far, we have relied on the reader's intuition to understand the meaning of the CCS operators. To make these meanings precise, CCS is equipped with an *operational semantics* that is intended precisely to define the execution steps that processes may engage in. This semantics is usually specified in the form of a ternary relation, $\longrightarrow$; intuitively, $p \xrightarrow{a} p'$ holds if system p is capable of engaging in action a and then behaving like p'. Process algebras such as CCS typically define $\longrightarrow$ inductively using a collection of *inference rules* for each operator. These rules have the following form.

$$\boxed{\frac{premises}{conclusion}\,(side\ condition)}$$

A rule states that, if one has established the premises, and the side condition holds, then one may infer the conclusion. This presentation style for operational semantics is often called *SOS*, for *Structural Operational Semantics*, and was devised by Plotkin [67].

The remainder of this section covers the SOS rules for CCS and shows how they may be used rigorously to characterize the behavior of CCS system descriptions. We group the rules on the basis of the CCS operators to which they apply.

nil. The CCS process *nil* has no rules and thus is incapable of any transitions.

Prefixing. The prefixing operator contains one rule.

$$\boxed{\frac{}{a.p \xrightarrow{a} p}}$$

This rule has no premises, and the conclusion states that processes of the form $a.p$ may engage in a and thereafter behave like p. Note that the side condition is omitted; in such cases it is assumed to be "true".

Choice. The choice operator has two symmetric rules.

$$\boxed{\frac{p \xrightarrow{a} p'}{p+q \xrightarrow{a} p'}} \quad \boxed{\frac{q \xrightarrow{a} q'}{p+q \xrightarrow{a} q'}}$$

These rules in essence state that a system of the form $p+q$ "inherits" the transitions of its subsystems p and q.

Parallel Composition. The parallel composition operator has three rules, the first two of which are symmetric.

$$\boxed{\frac{p \xrightarrow{a} p'}{p|q \xrightarrow{a} p'|q}} \quad \boxed{\frac{q \xrightarrow{a} q'}{p|q \xrightarrow{a} p|q'}}$$

These rules indicate that | interleaves the transitions of its subsystems. The next rule allows processes connected by | to interact.

$$\frac{p \xrightarrow{a} p', q \xrightarrow{\overline{a}} q'}{p|q \xrightarrow{\tau} p'|q'}$$

According to this rule, subsystems may *synchronize* on complementary actions (i.e., inputs and outputs on the same port). Note that the action produced as the result of the synchronization is a τ; since $\overline{\tau}$ is undefined, this ensures that synchronizations involve only two partners.

Restriction. The restriction operator has one rule.

$$\frac{p \xrightarrow{a} p'}{p\backslash L \xrightarrow{a} p'\backslash L}\ (a, \overline{a} \notin L)$$

This rule, which includes a side condition, only allows actions not mentioned in L (or whose complements are not in L) to be performed by $p\backslash L$. Restriction in effect "localizes" actions in L, since the operator forbids the system's environment from interacting with the system using them.

Relabeling. The relabeling operation has one rule.

$$\frac{p \xrightarrow{a} p'}{p[f] \xrightarrow{f(a)} p'[f]}$$

As the intuitive account above suggests, $p[f]$ engages in the same transitions as p, the difference being that the actions are relabeled via f.

Process Variables. The behavior of process variables is given by one rule.

$$\frac{p \xrightarrow{a} p'}{C \xrightarrow{a} p'}\ (C \stackrel{\Delta}{=} p)$$

This rule states that a system C behaves like the body, p, of its definition $C \stackrel{\Delta}{=} p$.

32.2.1.3 CCS and Labeled Transition Systems

The definition of $\longrightarrow$ just given allows CCS processes to be viewed as state machines of a certain type. To begin with, we show how CCS may be viewed as a structure called a *labeled transition system* consisting of a collection of possible system states and transitions.

Definition 1 A *labeled transition system* (LTS) is a triple $\langle Q, A, \longrightarrow \rangle$, where Q is a set of states, A is a set of actions, and $\longrightarrow \subseteq Q \times A \times Q$ is a transition relation.

Some definitions of LTS also designate a start state. We refer to labeled transition systems of this form (i.e., quadruples of the form $\langle Q, A, \longrightarrow, q_S \rangle$ where $q_S \in Q$ is the start state) as *rooted* labeled transition systems.

Perhaps surprisingly, the definitions in this chapter show that CCS may be viewed as a single LTS. Recall that $\mathscr{P}_{CCS}$ represents the (infinite) set of syntactically valid CCS system definitions, and let $\longrightarrow_{CCS}$ be the transition relation defined in the previous subsection. Then, $\langle \mathscr{P}_{CCS}, A_{CCS}, \longrightarrow_{CCS} \rangle$ satisfies the definition of an LTS. This observation has two consequences. The first is that certain definitions, such as those for behavioral equivalences and refinement orderings, may be given in a language-independent manner by defining them with respect to LTSs. The second consequence is that individual system descriptions may be converted into rooted LTSs. Mathematically, for any CCS system p, the quadruple $\langle \mathscr{P}_{CCS}, A_{CCS}, \longrightarrow_{CCS}, p \rangle$ constitutes a rooted LTS. As $\mathscr{P}_{CCS}$ is infinite this observation is only of theoretical interest until one observes that not every state in $\mathscr{P}_{CCS}$ is reachable from p via (sequences of) $\longrightarrow_{CCS}$ transitions. Consequently, we may instead define another LTS, M_p, consisting only of CCS terms reachable from p in this fashion. If M_p contains only finitely many states, then it may be analyzed using algorithms for manipulating finite-state machines, such as those presented later in this chapter.

32.2.1.4 Bisimulation

Process algebras rely on notions of behavioral (pre)congruence as a basis for system analysis. A *congruence* for an algebra is an equivalence relation (i.e., a relation that is reflexive, symmetric, and transitive) that also has the substitution property: equivalent systems may be used interchangeably inside any larger system.[1] Formally, we define a *context* $C[\,]$ to be a system description with a "hole", $[\,]$; given a system description p, then, $C[p]$ represents the system obtained by filling the hole with p. Then an equivalence $\approx$ is a congruence for a language if, whenever $p \approx q$, then $C[p] \approx C[q]$, for any context $C[\,]$ built using operators in the language. It should be noted that relations that are congruences for some languages are not congruences for others.

In this section, we study congruences for CCS with a view toward defining a relation that relates systems with respect to their observable behavior. In each case, we first define an equivalence relation on states in an arbitrary LTS; since CCS may be viewed as an LTS, these relations may then be used to relate CCS system descriptions. We then consider the suitability of the equivalence from the standpoint of the observable behavior to which it is sensitive and consider whether or not the relation is a congruence for CCS. In the first part of the section, we make no special allowance for the unobservability of the action τ, deferring its treatment until later.

When should two CCS processes be considered indistinguishable? The previous subsection showed how CCS terms may be converted into rooted labeled transition

[1] A precongruence is a preorder, i.e., a reflexive and transitive relation, that also has the substitution property.

systems (i.e., labeled transition systems with start states). This suggests an obvious notion of equivalence:[2] consider two CCS terms to be equivalent if they can perform the same sequences of actions. This definition would thus equate terms $a.(b.nil + c.nil)$ and $a.b.nil + a.c.nil$, since the two systems can each perform (all prefixes of) the sequences ab and ac. However, CCS is intended to model interactive systems that execute by engaging with their environments; from this perspective, these two systems can be seen as distinct. In particular, after an a action the system $a.(b.nil + c.nil)$ is in the state $b.nil + c.nil$: the choice of b or c is still available to the environment. The term $a.b.nil + a.c.nil$, on the other hand, has two possible successors after a: $b.nil$ and $c.nil$. In each case, one of the choices of b and c is no longer available. So while $a.(b.nil + c.nil)$ and $a.b.nil + a.c.nil$ are equated by this definition, the states they reach after an a-transition are not related by this definition.

This last observation suggests that an appropriate equivalence for CCS, and indeed for any language permitting the definition of nondeterministic systems, could have a "recursive" flavor: execution sequences for equivalent systems ought to pass through equivalent states. This intuition underlies the definition of *bisimulation*, or *strong* equivalence. The name of the equivalence stems from the fact that it is defined in terms of special relations called bisimulations.

Definition 2 Let $\langle Q, A, \longrightarrow \rangle$ be an LTS. A relation $R \subseteq Q \times Q$ is a *bisimulation* if, whenever $\langle p, q \rangle \in R$, the following conditions hold for any a.

1. if p' is such that $p \xrightarrow{a} p'$ then $q \xrightarrow{a} q'$ for some q' such that $\langle p', q' \rangle \in R$.
2. if q' is such that $q \xrightarrow{a} q'$ then $p \xrightarrow{a} p'$ for some p' such that $\langle p', q' \rangle \in R$.

Intuitively, if two systems are related by a bisimulation, then it is possible for each to simulate, or "track", the other's behavior: hence the term *bi*simulation. More specifically, for a relation to be a bisimulation, related states must be able to match each other's transitions by moving to related states. Two states are then *bisimulation equivalent* exactly when a bisimulation may be found relating them.

Definition 3 States p and q of some LTS are *bisimulation equivalent* (notation $p \sim q$), or *bisimilar*, if there exists a bisimulation R on that LTS containing $\langle p, q \rangle$.

Since CCS may be viewed as an LTS, one may use $\sim$ to relate CCS processes. As examples, we have the following.

1. $a.b.nil + a.b.nil \sim a.b.nil$,
2. $a.b.nil + a.c.nil \nsim a.(b.nil + c.nil)$.

Bisimulation is also considered in the chapter of the Handbook on *Abstraction and Abstraction-Refinement* (Chap. 13 [27]). There, bisimulation is defined over Kripke structures (with state labels) rather than LTSs (with edge labels).

[2] This equivalence relation is often called *trace equivalence*.

Bisimulation equivalence has a number of pleasing properties. Firstly, for any labeled transition system it is indeed an equivalence; that is, the relation $\sim$ is reflexive, symmetric, and transitive. Secondly, it can be shown in a precise sense that two equivalent systems must have the same "deadlock potential"; this point is addressed in more detail below. Thirdly, $\sim$ implies trace equivalence, and coincides with it if the LTS is *deterministic* in the sense that every state has at most one outgoing transition per action. Finally, $\sim$ is a congruence for CCS; if $p \sim q$ then p and q may be used interchangeably inside any larger system.

Bisimilarity does suffer from a flaw from the perspective of CCS and other process algebras allowing asynchronous execution: it is too sensitive to internal computation. In particular, the definition does not take account of the special status that τ has *vis-à-vis* other actions.[3] For example, the systems $a.\tau.b.nil$ and $a.b.nil$ are not bisimulation equivalent, even though an external observer cannot detect the difference between them. Nevertheless, $\sim$ has been studied extensively in the literature, and for process algebras in which internal computation in one component can indeed affect the behavior of other components, it is a reasonable basis for verification.

32.2.1.5 A Logical Characterization of Bisimilarity

The preceding discussion stated that $\sim$ relates systems on the basis of their relative "deadlock potentials". The remainder of this subsection makes this statement precise by defining a logic, called the Hennessy–Milner Logic (HML) [37], that permits the formulation of simple system properties, including potentials for deadlock. The logic also characterizes $\sim$ in the following sense: two systems are bisimilar if and only if they satisfy exactly the same formulae in the logic.

Syntax of HML

The definition of HML is parameterized with respect to a set A of actions. Given such a set, the syntax of HML formulae can be given via the following grammar.

$$\begin{aligned} \phi ::= & \; tt \\ | \; & \; ff \\ | \; & \; \phi \wedge \phi \\ | \; & \; \phi \vee \phi \\ | \; & \; \langle a \rangle \phi \\ | \; & \; [a]\phi \end{aligned}$$

We use Φ for the set of all well-formed HML formulae.

[3] Exactly the same can be said of the notions of trace equivalence and deterministic transition system given above: the former is usually, and the latter frequently, formulated differently to take account of the special role of τ. The definitions given above are, however, appropriate when studying this strong form of bisimilarity.

The constructs in the logic may be understood as follows. First, it should be noted that formulae are intended to be interpreted with respect to states in a labeled transition system. Then *tt* and *ff* represent the constants "true" and "false" that hold of any state and no state, respectively, while $\wedge$ and $\vee$ denote conjunction ("and") and disjunction ("or"), respectively. The final two operators are referred to as *modalities*, as they permit statements to be made about the transitions emanating from a state; thus HML is a *modal logic*. A state satisfies $\langle a\rangle\phi$ if a target state of one of its a-transitions satisfies ϕ, while $[a]\phi$ holds of a state if the target states of all of its a-transitions satisfy ϕ.

Semantics of HML

In order to formalize the meaning of HML, we first fix a labeled transition system $\mathscr{L} = \langle Q, A, \longrightarrow\rangle$ with the same action set as the logic. We then define a relation $\models_{\mathscr{L}} \subseteq Q \times \Phi$; intuitively, $q \models_{\mathscr{L}} \phi$ should hold if state q satisfies ϕ. The formal definition is given inductively as follows.

- $q \models_{\mathscr{L}} tt$ for any $q \in Q$.
- $q \models_{\mathscr{L}} ff$ for no $q \in Q$.
- $q \models_{\mathscr{L}} \phi_1 \wedge \phi_2$ if and only if $q \models_{\mathscr{L}} \phi_1$ and $q \models_{\mathscr{L}} \phi_2$.
- $q \models_{\mathscr{L}} \phi_1 \vee \phi_2$ if and only if $q \models_{\mathscr{L}} \phi_1$ or $q \models_{\mathscr{L}} \phi_2$.
- $q \models_{\mathscr{L}} \langle a\rangle\phi$ if and only if $q \stackrel{a}{\longrightarrow} q'$ and $q' \models_{\mathscr{L}} \phi$ for some $q' \in Q$.
- $q \models_{\mathscr{L}} [a]\phi$ if and only if for every q' such that $q \stackrel{a}{\longrightarrow} q'$, $q' \models_{\mathscr{L}} \phi$.

This definition includes some subtleties that deserve comment. To begin with, formula $[a]ff$ is satisfied by any state *not* having an a-transition; such states vacuously fulfil the requirement imposed by $[a]$. Indeed, a state with no a-transitions satisfies $[a]\phi$, for any ϕ. These facts also imply that a state incapable of any action in the set $\{a_1, \ldots, a_n\}$ will satisfy the formula $[a_1]ff \wedge \cdots \wedge [a_n]ff$. If such a state occurs in an environment that requires one of these actions, then a deadlock results. In a related vein, a state satisfies $\langle b\rangle tt$ if and only if it has a b-transition; more generally, given a (non-empty) sequence of actions $b_1 \ldots b_m$, a state includes $b_1 \ldots b_m$ as one of its *strong traces* if and only if the state satisfies the formula $\langle b_1\rangle \cdots \langle b_m\rangle tt$. Finally, consider a state satisfying a formula of the form

$$\langle b_1\rangle \cdots \langle b_m\rangle\big([a_1]ff \wedge \cdots \wedge [a_n]ff\big).$$

Such a state satisfies this formula if it can engage in the sequence $b_1 \ldots b_m$ and arrive at a state that rejects offers for interaction involving any of $a_1, \ldots, a_n$. In an environment capable of exercising the sequence $b_1 \ldots b_m$ and then requiring an interaction involving one of $a_1, \ldots, a_n$, the given state could deadlock. It is in this sense that HML permits the formulation of properties expressing potentials for deadlock.

Bisimulation *vis-à-vis* HML

The relationship between HML and $\sim$ is captured by the following theorem, which states that HML characterizes $\sim$ for labeled transition systems that are *image-finite*.

An LTS is image-finite if every state in the LTS has at most finitely many transitions sharing the same action label. In practice almost all labeled transition systems satisfy this requirement; in particular, CCS does, provided the definitions of process variables obey a syntactic restriction ("guardedness").

Theorem 1 *Let $\mathscr{L} = \langle Q, A, \longrightarrow \rangle$ be an image-finite LTS, and let $p, q \in Q$. Then $p \sim q$ if and only if for all HML formulae ϕ, either $p \models_{\mathscr{L}} \phi$ and $q \models_{\mathscr{L}} \phi$, or $p \not\models_{\mathscr{L}} \phi$ and $q \not\models_{\mathscr{L}} \phi$.*

On the one hand, this result and the previous discussion substantiate the claim that bisimulation equivalence requires equivalent systems to have the same "deadlock potentials". On the other hand, the theorem provides a useful mechanism for explaining why two systems fail to be equivalent; one need only present a formula satisfied by one system and not the other. For example, consider the system p given by $a.(b.nil + c.nil)$ and the system q given by $a.b.nil + a.c.nil$. Since $p \nsim q$ there must be a formula satisfied by one and not the other. One such formula is $[a]\langle b\rangle tt$, which is satisfied by p but not by q.

32.2.1.6 Observational Equivalence and Congruence for CCS

This subsection presents a coarsening of bisimulation equivalence that is intended to relax its sensitivity to internal computation. The definition of this relation relies on the introduction of so-called *weak* transitions.

Definition 4 Let $\langle Q, A, \longrightarrow \rangle$ be an LTS with $\tau \in A$, and let $q \in Q$.

1. If $s \in A^*$ then $\hat{s} \in (A - \{\tau\})^*$ is the action sequence obtained by deleting all occurrences of τ from s.
2. Let $s \in (A - \{\tau\})^*$. Then $q \stackrel{s}{\Longrightarrow} q'$ iff there exists s' such that $q \stackrel{s'}{\longrightarrow} q'$ and $s = \hat{s'}$.

Intuitively, $\hat{s}$ returns the "visible content" (i.e., non-τ elements) of sequence s; in particular, if $a \in A$ then $\hat{a} = \varepsilon$ if $a = \tau$, while $\hat{a} = a$ if $a \neq \tau$. In addition, $q \stackrel{s}{\Longrightarrow} q'$ if q can perform a sequence of transitions with the same visible content as s and evolve to q'. In this case, note that the sequence of transitions that is performed is the same as s except that it potentially includes an arbitrary number of τ-transitions among the visible actions of s. In particular, $q \stackrel{\varepsilon}{\Longrightarrow} q'$ if a sequence of τ-transitions leads from q to q', while for a single visible action a, $q \stackrel{a}{\Longrightarrow} q'$ if q can perform an a, possibly preceded and followed by some internal computation, to arrive at q'.

We may now define *weak bisimulations* as follows.

Definition 5 Let $\langle Q, A, \longrightarrow \rangle$ be an LTS, with $\tau \in A$. Then a relation $R \subseteq Q \times Q$ is a *weak bisimulation* if, whenever $\langle p, q \rangle \in R$, the following hold for all $a \in A$ and $p', q' \in Q$.

1. If $p \xrightarrow{a} p'$ then $q \stackrel{\hat{a}}{\Longrightarrow} q'$ for some q' such that $\langle p', q' \rangle \in R$.
2. If $q \xrightarrow{a} q'$ then $p \stackrel{\hat{a}}{\Longrightarrow} p'$ for some p' such that $\langle p', q' \rangle \in R$.

States p and q are *observationally equivalent*, or *weakly equivalent*, or *weakly bisimilar*, if there exists a weak bisimulation R containing $\langle p, q \rangle$. When this is the case, we write $p \approx q$.

A weak bisimulation closely resembles a regular bisimulation; the only difference lies in the fact that systems may use weak transitions to simulate normal transitions in the other system.

As CCS is a labeled transition system whose action set contains τ, the definition of $\approx$ may be used to relate CCS system descriptions. Doing so leads to the following observations.

- $a.\tau.b.nil \approx a.b.nil$.
- For any p, $\tau.p \approx p$.

Even though it ignores internal computation, observational equivalence still enjoys a similar degree of deadlock-sensitivity to bisimulation equivalence: a variant of HML can be defined that characterizes $\approx$ in the same way that HML characterizes $\sim$. (This logic replaces the $\langle a \rangle$ and $[a]$ modalities of HML by two new operators, $\langle\langle a \rangle\rangle$ and $[[a]]$; a state $q \models_{\mathscr{L}} \langle\langle a \rangle\rangle \phi$ if there exists a q' such that $q \stackrel{\hat{a}}{\Longrightarrow} q'$ and $q' \models_{\mathscr{L}} \phi$, and similarly for $[[a]]$.) Consequently, it would appear to be a viable candidate for relating CCS system descriptions. Unfortunately, however, it is *not* a congruence for CCS. To see why, consider the context $C[\,]$ given by $[\,] + b.nil$. It is easy to establish that $p \approx q$, where p is given by $\tau.a.nil$ and q by $a.nil$. However, $C[p] \not\approx C[q]$. To see this, note that $C[p] \xrightarrow{\tau} a.nil$. This transition must be matched by a weak ε-labeled transition from $C[q]$. The only such transition $C[q]$ has is $C[q] \stackrel{\varepsilon}{\Longrightarrow} C[q]$. However, $a.nil \not\approx C[q]$, since the latter can engage in a b-labeled transition that cannot be matched by the former.

This defect of $\approx$ arises from the interplay between $+$ and the initial internal computation that a system might engage in; in particular, the only CCS operator that breaks the congruence-hood of $\approx$ is $+$. Some researchers reasonably suggest that this is an argument against including $+$ in the language. Milner [58, 59] adopts another point of view that we pursue in the remainder of this section, and that is to focus on finding the *largest* CCS congruence $\approx^C$ that implies $\approx$. Such a largest congruence is guaranteed to exist [37].

Definition 6 Let $\langle Q, A, \longrightarrow \rangle$ be an LTS with $\tau \in A$, and let $p, q \in Q$. Then $p \approx^C q$ if the following hold for all $a \in A$ and $p', q' \in Q$.

1. If $p \xrightarrow{a} p'$ then $q \stackrel{a}{\Longrightarrow} q'$ for some q' such that $p' \approx q'$.
2. If $q \xrightarrow{a} q'$ then $p \stackrel{a}{\Longrightarrow} p'$ for some p' such that $p' \approx q'$.

Some remarks about this relation are in order. Firstly, it should be noted that for $p \approx^C q$ to hold, any τ-transition of p must be matched by a $\stackrel{\tau}{\Longrightarrow}$-transition

of q; in particular, this weak transition must consist of a *non-empty* sequence of τ-transitions. Secondly, the definition is not recursive: the targets of initial matching transitions need only be related by $\approx$. Finally, it indeed turns out that $\approx^C$ is a congruence for CCS and that it is the largest CCS congruence entailing $\approx$. That is, $p \approx^C q$ implies $p \approx q$, and for any other congruence R such that $p\,R\,q$ implies $p \approx q$, $p\,R\,q$ also implies $p \approx^C q$. As examples, we have the following.

1. $a.\tau.b.nil \approx^C a.b.nil$.
2. $\tau.a.nil \not\approx^C a.nil$, since the $\xrightarrow{\tau}$ transition of the former cannot be matched by a $\stackrel{\tau}{\Longrightarrow}$ transition of the latter.
3. For any p, q, if $p \approx q$ then $\tau.p \approx^C \tau.q$.

32.2.1.7 Axiomatizations of Bisimilarity and Observational Congruence

Process algebras often include (in)equational proof systems for checking whether two terms are related by the behavioral congruences/precongruences defined for them. The proof systems include a fixed set of proof rules, together with a number of axioms specifying equality/ordering relationships involving the different operators in the algebra. This subsection gives the proof systems for bisimulation equivalence and observational congruence for *finite CCS*, namely, CCS without process constants.

Inference Rules for Equational Reasoning

The inference rules used in equational reasoning are given in Fig. 1. These rules are intended to prove judgements of the form $t_1 = t_2$, where t_1 and t_2 are terms in a given algebra. Each rule consists of three parts: a list of *premises*, given above the line in the rule; a *conclusion*, given below the rule; and a name (only included for explanatory purposes). In the figure, Rule (R) ("reflexivity") has no premises; it asserts that every term is equal to itself. Rule (C) ("commutativity") says that if a term is equal to a second, then the second is equal to the first. Rule (T) ("transitivity") captures the well-known transitive property of equality. Finally, Rule (S) ("substitutivity") indicates that if two terms have been proven equal, then one may be substituted for another inside any larger term, represented here as the surrounding context.

Stylistically, equational proofs are usually written in the following form:

$$\begin{aligned} t_1 &= t_2 \\ &= t_3 \\ &\vdots \\ &= t_n. \end{aligned}$$

This is a proof that term $t_1 = t_n$. The inference rules are applied implicitly, rather than explicitly.

$$\frac{-}{x=x}\,(R) \quad \frac{x=y}{y=x}\,(C) \quad \frac{x=y \;\; y=z}{x=z}\,(T) \quad \frac{x=y}{C[x]=C[y]}\,(S)$$

Fig. 1 Rules of equational reasoning

Table 1 Equational axioms for bisimulation equivalence

(A1)	$P+Q=Q+P$
(A2)	$P+(Q+R)=(P+Q)+R$
(A3)	$P+nil=P$
(A4)	$P+P=P$
(Exp)	Let $P\equiv\sum_{i\in I}a_i.P_i$, $Q\equiv\sum_{j\in J}b_j.Q_j$. Then: $P\mid Q=\sum_{i\in I}a_i.(P_i\mid Q)$ $\quad+\sum_{j\in J}b_j.(P\mid Q_j)$ $\quad+\sum_{\langle i,j\rangle\in\{\langle i,j\rangle\in I\times J\mid a_i=\overline{b_j}\}}\tau.(P_i\mid Q_j)$
(Res1)	$nil\backslash L=nil$
(Res2)	$(a.P)\backslash L=\begin{cases}nil & \text{if } a\in L \text{ or } \overline{a}\in L\\ a.(P\backslash L) & \text{otherwise}\end{cases}$
(Res3)	$(P+Q)\backslash L=(P\backslash L)+(Q\backslash L)$
(Rel1)	$nil[f]=nil$
(Rel2)	$(a.P)[f]=f(a).(P[f])$
(Rel3)	$(P+Q)[f]=(P[f])+(Q[f])$

Axiomatizing Bisimilarity

Given the equational reasoning rules given above, the remaining step in specifying an equational proof system for a given algebra and congruence relation is to give a set of *non-logical* (i.e., algebra-specific) axioms specifying the desired equalities that hold among the operators. Table 1 gives these for finite CCS and bisimilarity.

The presentation of the equational axioms is by operator. Laws (A1)–(A4) involve $+$ and *nil*; they establish that the former is commutative (A1), associative (A2) and idempotent (A4), and that *nil* is a unit for $+$ (A3). The (Exp) axiom shows how instances of $\mid$ at the top level of a term may be pushed inside the term, provided the terms to which $\mid$ is applied are in a certain normal form. Strictly speaking, (Exp) is an axiom schema, rather than a single axiom; that is, it is shorthand for an infinite number of axioms. The laws (Res1)–(Res3) and (Rel1)–(Rel3) describe the interactions between restriction (respectively, relabeling), and $a.$, *nil*, and $+$. One observation about these laws is that any finite CCS term can be rewritten using them into another term involving only $a.$, *nil*, and $+$.

This proof system can be shown to be *sound* and *complete* for bisimulation equivalence. That is, whenever one can prove $p=q$ for specific finite-CCS terms, then $p\sim q$ is guaranteed to hold (soundness). Conversely, if p,q are finite-CCS terms such that $p\sim q$, then one can prove that $p=q$ (completeness). What follows is

Table 2 The CCS τ laws

$a.\tau.P = a.P$	$(\tau 1)$
$P + \tau.P = \tau.P$	$(\tau 2)$
$a.(P + \tau.Q) = a.(P + \tau.Q) + a.Q$	$(\tau 3)$

an example equational proof; the names of the axioms are given in the rightmost column. Here we write 0 for *nil*.

$$\begin{aligned} a.(b.0 + (c.0 + b.0)) + 0 &= a.(b.0 + (c.0 + b.0)) \quad \text{(A3)} \\ &= a.(b.0 + (b.0 + c.0)) \quad \text{(A1)} \\ &= a.((b.0 + b.0) + c.0) \quad \text{(A2)} \\ &= a.(b.0 + c.0). \quad \text{(A4)} \end{aligned}$$

Axiomatizing Observational Congruence

An axiomatization for $\approx^C$ can also be given for finite CCS. It should first be noted that bisimilarity implies observational congruence; that is, if $p \sim q$ then $p \approx^C q$. This implies that all the non-logical axioms given in Table 1 are sound also for $\approx^C$. To those rules we can add the ones given in Table 2 in order to obtain a sound and complete axiomatization for $\approx^C$. (These new axioms are sometimes referred to as the τ *laws*.)

It is tempting to replace the three τ laws by a single law, $\tau.P = P$. However, this law is not sound for $\approx^C$, as the previous discussion showed: $\tau.a.nil \not\approx^C a.nil$.

Axiomatizing Full CCS

The previous equational axiomatizations treated only finite CCS, namely, CCS without process definitions. One is led naturally to the question of axiomatizing full CCS, including terms with process constants. Unfortunately, such a full axiomatization cannot be given; both $\sim$ and $\approx^C$ are not recursively enumerable for full CCS, and thus no proof system can be given that is both sound and complete. Researchers have instead focused on giving sound and complete axiomatizations for fragments of CCS that include process constants (cf. *regular* CCS), and on studying sound, but not complete, approaches to reasoning about full CCS with process constants (cf. the Unique Fixpoint Induction rule). The interested reader is encouraged to consult the CCS-specific references for more information on these topics.

32.2.2 CSP: Process Algebra via Denotational Semantics

In the 1970s, Hoare developed the Communicating Sequential Processes (CSP) programming model for concurrent, message-passing systems. Initially, this was an

imperative programming language [40] combining primitives for sending and receiving messages in a handshaking (AKA synchronous) manner and top-level parallelism.

To understand the foundational semantic issues of this model of concurrency, Hoare, assisted by his then students Brookes and Roscoe, developed the process algebra version of CSP, sometimes called TCSP or Theoretical CSP, in the late 1970s and early 1980s. The primary references for the latter development are [12, 13, 41, 42]. Like CCS and ACP (see Sect. 32.2.3 below), CSP includes a small number of basic constructs for assembling systems out of subsystems. The operators differ in several respects from those of CCS. A more fundamental distinction, however, lies in the approach taken to define the semantics of CSP. Instead of the operational style favored by CCS, (T)CSP was first equipped with *denotational* semantics. Specifically, a mathematical set of semantic objects was developed, and the CSP operators given formal specifications as functions over this set. The refinement ordering and equivalence on process terms are then inherited from the underlying ordering of this set. The additional structure in the set ("model") ensures a coherent treatment of recursively defined processes, though this, like the abstract operator definitions, has to be reconciled with operational intuition.

Later, the CSP process algebra was combined with a pure functional programming language in the CSP_M language [80] that is used in automated analysis, once again making it an expressive programming language, this time declarative.

The rest of this subsection elaborates on the ideas above. The syntax of (a version of) CSP is introduced, and the considerations underlying the design of the semantic model considered, namely *failures-divergences*. Then, the semantics of CSP is given, and the semantic refinement ordering discussed. Finally, connections with an operational semantics of CSP are described.

32.2.2.1 CSP Syntax

In common with CCS, CSP builds process descriptions from atomic actions using a set of operators, here slightly larger than that of CCS.

Actions in CSP

All actions that appear in the CSP language represent visible synchronizations that require the agreement of the process and its environment to occur.

No particular structure is assumed of CSP's events, though compound events are frequently constructed using an infix dot, typically with a label followed by zero or more data components as in $output.2.3.true$. CSP_M mandates this form, where for each label (generally called a *channel* when there are data components attached) the number and types of the data components are declared and therefore fixed. The set of all such events usable in the language is generally written Σ. Depending on the class of processes being described, Σ might be finite or infinite.

In CSP, there is no pairing of events as in the α, $\overline{\alpha}$ of CCS. Instead processes typically synchronize multiple (perhaps more than two) copies of the same event.

Apart from this main sort of action, there are two other events to consider. Firstly, our intuition about the operational behavior of CSP processes includes the same τ used in CCS, but τ plays no part in the language itself. Secondly, there is a special visible event $\checkmark$, introduced only via the process $SKIP$, representing successful termination; so, when it appears, it is always the last event a process performs. τ and $\checkmark$ are not members of Σ.

CSP Constructs

- $STOP$ represents a process that can perform no action.
- Given $a \in \Sigma$, the *prefixing operator* $a \rightarrow$ allows an action to be prepended onto an existing system description and corresponds precisely to prefixing in CCS. CSP allows us to generalize prefixing beyond simple actions with choice forms such as $?x : A \rightarrow$ and $c?x \rightarrow$, which respectively allow the process to communicate any event from the set $A \subseteq \Sigma$ and any event on the channel c. In such cases, the process term following the prefix can depend on the identifier x.
- $\sqcap$ represents internal, *nondeterministic choice*. The system $P_1 \sqcap P_2$ has the choice of behaving like P_1 or P_2 as it wishes: other processes and the environment have no influence over this choice.

  $\sqcap S$, for a nonempty set S of processes, is the nondeterministic choice over them all. In general, S can be infinite, but this introduces infinite, or unbounded, nondeterminism.
- $\Box$ represents *external choice*. $P_1 \Box P_2$ allows its environment all actions from Σ initially offered by P_1 or P_2, and behaves like one of these processes does after the chosen action. So if the chosen action a is only possible for P_1, the process's subsequent behavior is that of P_1 after a. If the chosen a is initially possible for both P_1 and P_2, the subsequent behavior is nondeterministically (in the sense of $\sqcap$) like that of P_1 or P_2 after a. Importantly (and unlike CCS), $P_1 \Box P_2$ is not resolved by either process performing an invisible τ.
- $\|$ denotes *parallel composition*. This simply synchronizes events which are common between its arguments, letting them remain visible for further synchronization. There are two variants on it: $P_A \|_B Q$ (alphabetized parallel) allows P to perform actions in A and Q in B, with events in $A \cap B$ being synchronized (i.e., such an action only occurs when both P and Q perform it). $P \underset{A}{\|} Q$ (generalized or interface parallel), allows each of P and Q to perform actions outside A independently, even though there may be such events common to P and Q. Just the events in A are synchronized.

  Interleaving parallel $P \,|||\, Q$, is a synonym for $P \underset{\emptyset}{\|} Q$, and allows P and Q to communicate freely and independently.
- $\setminus A$ denotes *hiding*. $P \setminus A$ behaves like P except that all actions in $A \subseteq \Sigma$ become invisible τs. Unlike CCS restriction, it does not prevent the actions in A, but rather prevents the environment either seeing them or blocking them.

- $[\![R]\!]$, for R a relation on Σ, is *renaming*. If P can perform the action a and $a\,R\,b$ then $P[\![R]\!]$ can perform b. Conventionally, R is total on the actions possible for P, but is not restricted to be a function or injective.

For the purposes of this summary presentation, we will describe the semantics of only *core CSP* built from the operators in the above list (using only simple prefix, and only the interface form of parallel $\underset{X}{\|}$). Just about all presentations of CSP, however, include the rest of the constructs above together with some or all of the following constructs, each of which provides means for one process to hand control over to a second one.

- $SKIP$ represents a process that *terminates successfully* by performing the event $\checkmark$. The counterpart to this is *sequential composition* $P; Q$, which starts Q when P terminates.
- $P \Delta Q$ is the *interrupt* operator, in which P runs but if Q performs an action, it takes over from P; so, $(a \to P)\Delta(b \to Q) = (a \to (P \Delta (b \to Q)))\Box(b \to Q)$.
- $P\,\Theta_a\,Q$ allows P to run, but as soon as P performs a, Q takes over. Thus $(b \to a \to P)\,\Theta_a\,Q = b \to a \to Q$. This operator models the left-hand argument throwing an exception, via a, and is therefore called the *throw* operator.

The extended set of operators can be important for practical applications, and is important for a number of theoretical results about the classification of CSP models and one we allude to later about the expressiveness of CSP.

Like CCS, CSP has an operational semantics over labeled transition systems. Unlike CCS, this was not the primary intuition behind the language, but is now used extensively in tools such as FDR, and provides a reference for checking the accuracy of other semantics. For example, the SOS rules for $\Box$ and $\underset{X}{\|}$ are as follows (note how they reflect the intuition above and contrast with the corresponding CCS clauses):

$$\frac{p \xrightarrow{\tau} p'}{p\Box q \xrightarrow{\tau} p'\Box q} \qquad \frac{q \xrightarrow{\tau} q'}{p\Box q \xrightarrow{\tau} p\Box q'}$$

$$\frac{p \xrightarrow{a} p'}{p\Box q \xrightarrow{a} p'}\ a \neq \tau \qquad \frac{q \xrightarrow{a} q'}{p\Box q \xrightarrow{a} q'}\ a \neq \tau$$

$$\frac{p \xrightarrow{x} p'}{p \underset{X}{\|} q \xrightarrow{x} p' \underset{X}{\|} q}\ x \notin X \qquad \frac{q \xrightarrow{x} q'}{p \underset{X}{\|} q \xrightarrow{x} p \underset{X}{\|} q'}\ x \notin X$$

$$\frac{p \xrightarrow{a} p', q \xrightarrow{a} q'}{p \underset{X}{\|} q \xrightarrow{a} p' \underset{X}{\|} q'}\ a \in X$$

The rest of the clauses may be found in [75], where there is also a characterization of the *CSP-like* style of operational semantics that guarantees a language has a semantics in CSP models. This might apply to any language where the operational semantics, like those of the notations in this chapter, are cast in visible and

τ actions, and given in terms of structural operational semantics. In a sense made precise there, a CSP-like operational semantics is one where there are no negative premises to actions, no copying of a process in flight, and where τ actions of active sub-processes are always enabled but have no effect on outer context.

It turns out that CCS is not CSP-like in this sense because of the way that τ interacts with $+$, which breaches the last of these conditions. However, versions of the π-calculus such as [79], where use of $+$ is confined to a limited class of terms, are CSP-like [74]. Any CSP-like language can (Chap. 9 of [75]) be translated in a semantics-preserving sense (strong bisimilarity in the absence of the termination event $\checkmark$) into the full (rather than core) CSP notation.

32.2.2.2 Models for CSP Processes

The development of the CSP process algebra was to some extent guided by the sort of operational intuition seen above for CCS, but at least as much by the quest for the right set of algebraic laws, as with ACP (see Sect. 32.2.3 below). However the core semantic understanding of CSP during its development was in terms of mathematical models based on observing the behavior of processes. Thus a process would be identified with the set(s) of all behaviors of some type or types that it is capable of performing. The touchstone of such models was that the semantics of CSP over them would satisfy the desired laws and offer a denotational semantics for the language at an appropriate level of abstraction. So such a model would necessarily be a congruence for the language, namely, it is possible to compute the value in a model of a process such as $P \Box Q$ in which one or more processes are combined by CSP operators, solely from the values of the components (here P and Q). It must also offer a technique for computing the fixed point represented by any recursive term. While CSP now has the SOS-style operational semantics described above, this was developed several years later than the earliest of these behavioral models.

Quite a number of such models have been developed, as set out in [75], but here we will concentrate on the core one originally set out in [13] as a development from the one of [12]. This is the *failures-divergences model*, and is the one after which FDR is named. In the presentation here, we will ignore the termination event $\checkmark$, since that adds slight complications and is not required for our core CSP. The following types of behavior make up the model:

- A *trace* is a sequence of events in Σ that the process can be observed to perform in order. In general, a trace can be finite or infinite, but in this model we only consider finite traces: members of Σ^*.[4]

[4]Traces are written $\langle a, b, c \rangle$ ($\langle\rangle$ being the empty trace), with $s \hat{} t$ being concatenation, $s \upharpoonright X$ for $X \subseteq \Sigma$ being restriction—the subsequence of s consisting of members of X—and $s \setminus X = s \upharpoonright (\Sigma \setminus X)$. We write $s \leq t$ if s is a prefix of t, namely $\exists s'.s \hat{} s' = t$. A^* is the set of all finite sequences whose members are drawn from the set A.

- A process *diverges* if it performs an infinite sequence of τ actions, thus never becoming *stable*. This might happen because of an infinite series of hidden actions, but also might represent a process that must unwind a recursion infinitely without ever reaching a visible action or even a state[5] like $STOP$. Note that τs are not included in these traces.
- A process *refuses* $X \subseteq \Sigma$ just when it is in a stable (τ-free) state V that cannot perform any member of X (i.e., $\nexists x \in X \cup \{\tau\}.V \xrightarrow{x}$)
- A *failure* (s, X) is the coupling of a trace s with a set X that a process can refuse after performing s.

A model is *divergence-strict* if it identifies all processes that can diverge before performing any visible action. So for example, if **div** is a process that simply diverges, such as $AS \setminus \{a\}$ for $AS = a \rightarrow AS$, a divergence-strict model will identify **div**, $\mathbf{div} \Box P$, $\mathbf{div} \sqcap P$, and $\mathbf{div}_X \|_Y P$ for any P, X, and Y since they can all perform an infinite unbroken sequence of τ actions from their initial states. However since $a \rightarrow \mathbf{div}$ does not diverge until after a visible action, there such models need not (and will not, except in the trivial model that identifies all processes) identify this last process with **div**. The failures-divergences model, which is divergence-strict, represents a process P as a pair of sets (F, D), where

- F is the set of failures it can perform, extended by all $(s\hat{}t, X)$ for divergences s of P as defined below. This divergence-strict set of failures is denoted $failures_\perp(P)$.
- D is the set of all extensions of traces after which P can diverge; i.e., the set of all $s\hat{}t$ such that P can perform the trace s and reach a divergent state. This divergence-strict set of divergences is denoted $divergences(P)$.

The reason for the assumption of divergence-strictness here is to make the model compositional. It can mainly be dispensed with, as shown in [73], but not without introducing an extra component of infinite traces (so that the representation of a process becomes $(F^\sharp, D^\sharp, I^\sharp)$ where $F^\sharp$ and $D^\sharp$ are failures and divergences without closure under divergence-strictness, and $I^\sharp$ is the set of all the process's infinite traces plus all infinite traces that have an infinite number of prefixes in $D^\sharp$, a weak variant on divergence-strictness.[6] There is also significant extra complexity in the calculation of the correct fixed point to denote the semantics of recursive terms.

The failures-divergences model itself, which we will write as $\mathcal{N}$, consists of all those pairs (F, D) satisfying the following healthiness conditions:

F1 $T = \{s \mid (s, X) \in F\}$ is non-empty and prefix-closed.

F2 $(s, X) \in F \wedge Y \subseteq X \Rightarrow (s, Y) \in F$. When a process refuses X, it refuses any subset of X.

[5] In discussing CSP, we generally identify a "process" with the corresponding element of the abstract semantic model under consideration. A "state", on the other hand, is a process term considered as part of the operational semantics.

[6] This inclusion of infinite traces where divergence can occur at infinitely many points is all that remains of divergence-strictness. [73] demonstrates that there can be no satisfactory fixed-point theory for recursion without it.

F3 $(s, X) \in F \wedge Y \cap \{x \mid s\hat{}\langle x\rangle \in T\} = \emptyset \Rightarrow (s, X \cup Y) \in F$. When a process refuses X it also refuses all events that the process can *never* perform after the same trace.
D1 $s \in D \wedge t \in \Sigma^* \Rightarrow s\hat{}t \in D$.
D2 $s \in D \Rightarrow (s, X) \in F$.

$\mathcal{N}$ is a complete partial order under the refinement order $\sqsubseteq_{FD}$ defined

$$(F, D) \sqsubseteq_{FD} (F', D') \equiv F \supseteq F' \wedge D \supseteq D'$$

whose bottom element is $(\Sigma^* \times \mathcal{P}(\Sigma), \Sigma^*)$, the value that represents any process that can diverge immediately. It has many maximal elements, representing *deterministic* processes: ones that can never diverge, and can never have the choice of accepting or refusing any event. If $SKIP$ and sequential composition are added to the language, the model becomes slightly more complex to define, because of the role of $\checkmark$ as a final signal event representing termination, and uses several extra healthiness conditions.

The two best-known alternatives to $\mathcal{N}$ are the *traces model*, which records only a process's finite traces, and the *stable failures model*, which records both finite traces and the failures not extended by divergence strictness. However there are many others, and the reader is referred to Chaps. 10–12 of [75] for a classification of those based on linear observations consisting of events, refusals, and their extension acceptances or *ready sets* that allow the observer to see the exact set of actions offered by a stable state. (Here, divergence can be viewed as the indefinite absence of stability.)

Rather than use a separate logic to specify properties of processes, the most common approach is to use the idea of *refinement*: in any CSP model, one process refines another if its recorded behavior is (component-wise where appropriate) as in the definition of $\sqsubseteq_{FD}$ above. Thus, restricting the range of a process's recorded behavior makes it more refined.

As well as having the obvious role of deciding when one proposed implementation is more refined than another, refinement is very frequently used in the form $Spec \sqsubseteq Imp$, where $Spec$ is a process designed to represent a specification, Imp is our proposed implementation, and the model used for $\sqsubseteq$ is chosen (in conjunction with $Spec$) so that it captures all of the behavior required to capture the specification in hand. Thus, a property that bans particular members of Σ, or one that says that events never happen in the wrong order, would use *trace* refinement; to check deadlock freedom you would use (stable) *failures* refinement to check that the process had no failure of the form (s, Σ) (i.e., reaching a state where no event is possible at all); but to check that a process will definitely respond when offered a set of events (i.e., can neither diverge nor deadlock when offered it) requires refinement over $\mathcal{N}$.

A *behavioral specification* cast in such a model is one that states that every observed behavior of the process P satisfies a given property. Thus, Hoare defined that a process whose alphabet consists of $\{in.x, out.x \mid x \in T\}$ is a partially correct buffer if the sequence of outputs on out is always a prefix of the inputs on in. This can be extended to a complete specification of a buffer over $\mathcal{N}$: here we can insist that the process never diverges, never refuses to output when non-empty and never refuses to input when non-full. In [42], Hoare cast such specifications in predicate

calculus, but it is always possible over any CSP model to find, for every behavioral specification R, a (possibly infinitary) *characteristic* process $Spec_R$ that both satisfies R and such that $Spec_R \sqsubseteq P$ if and only if P satisfies R.

Each CSP model defines its own equivalence on processes. So, for example, P and Q are failures-divergences equivalent if their representations in $\mathcal{N}$ are the same. Every model in the class considered in [75] gives a coarser equivalence than strong bisimulation, but all except the coarsest of them all (traces) is incomparable with weak bisimulation. However [75] the equivalence induced by every CSP model is coarser than the divergence-respecting weak bisimulation—the maximal weak bisimulation that never identifies a divergent node and a non-divergent node.

32.2.2.3 A Denotational Semantics for CSP

Given a model $\mathcal{M}$, any CSP term P represents a function from $\mathcal{M}$-environments (mappings from the free process identifiers to $\mathcal{M}$) to $\mathcal{M}$. Such functions can be calculated in terms of a denotational semantics for CSP over $\mathcal{M}$, consisting of a definition for each operator over $\mathcal{M}$, and a mechanism for finding the semantics of recursive terms.

This semantics for operators must coincide with operational intuition, and the fixed-point theory that delivers the meaning of recursions has to give the operationally correct values. Given that we have the operational semantics, these two statements can be tested mathematically by building congruence results, as discussed and demonstrated in the literature, for example [71, 72, 75].

As an example, we give here the semantics of the core language over $\mathcal{N}$. This is valid only for *finitely nondeterministic CSP*, namely without operators that can introduce infinite branching on any single action: these are infinite nondeterministic choice, hiding an infinite subset of Σ, and a renaming relation that maps infinitely many actions onto a single result. To handle unbounded nondeterminism (which is not relevant at present for model checking) one can add an extra model component of the process's divergence-closed infinite traces.[7]

The semantics below is that of [72] simplified by the removal of special cases for $\checkmark$. Note that a number of the definitions require divergence-strictness to be enforced by a special term, for example the last components of the unions in the definitions of $failures_\perp(P \Box Q)$ and $failures_\perp(P \underset{X}{\parallel} Q)$ below.

$$failures_\perp(STOP) = \{(\langle\rangle, X) \mid X \subseteq \Sigma\},$$
$$divergences(STOP) = \emptyset.$$
$$failures_\perp(a \to P) = \{(\langle\rangle, X) \mid a \notin X\} \cup \{(\langle a\rangle^\frown s, X) \mid (s, X) \in failures_\perp(P)\},$$
$$divergences(a \to P) = \{\langle a\rangle^\frown s \mid s \in divergences(P)\},$$
$$failures_\perp(P \sqcap Q) = failures_\perp(P) \cup failures_\perp(Q),$$
$$divergences(P \sqcap Q) = divergences(P) \cup divergences(Q),$$

[7] This fully divergence-strict model with infinite traces is coarser than the one alluded to earlier with only a weak form of divergence-strictness.

$$\begin{aligned}
failures_\perp(P \Box Q) = {} & \{(\langle\rangle, X) \mid (\langle\rangle, X) \in failures_\perp(P) \cap failures_\perp(Q)\} \\
& \cup \{(s, X) \mid (s, X) \in failures_\perp(P) \\
& \cup failures_\perp(Q) \wedge s \neq \langle\rangle\} \\
& \cup \{(\langle\rangle, X) \mid \langle\rangle \in divergences(P) \cup divergences(Q)\},
\end{aligned}$$

$$divergences(P \Box Q) = divergences(P) \cup divergences(Q),$$

$$\begin{aligned}
failures_\perp(P \underset{X}{\parallel} Q) = {} & \{(u, Y \cup Z) \mid Y \setminus X = Z \setminus X \\
& \wedge \exists s, t.(s, Y) \in failures_\perp(P) \wedge (t, Z) \in failures_\perp(Q) \\
& \qquad \wedge u \in s \underset{X}{\parallel} t\} \\
& \cup \{(u, Y) \mid u \in divergences(P \underset{X}{\parallel} Q)\},
\end{aligned}$$

$$\begin{aligned}
divergences(P \underset{X}{\parallel} Q) = {} & \{u \hat{} v \mid \exists s \in traces_\perp(P), t \in traces_\perp(Q). \\
& \quad u \in s \underset{X}{\parallel} t \wedge (s \in divergences(P) \\
& \quad \vee t \in divergences(Q))\}.
\end{aligned}$$

Here, $s \underset{X}{\parallel} t$ is the set of traces (empty when $s \restriction X \neq t \restriction X$) where there is a labeling of all events outside X with 1 or 2 such that deleting all events labeled 1 gives t, and deleting all events labeled 2 gives s.

$$\begin{aligned}
failures_\perp(P \setminus X) = {} & \{(s \setminus X, Y) \mid (s, Y \cup X) \in failures_\perp(P)\} \\
& \cup \{(s, Y) \mid s \in divergences(P \setminus X)\}.
\end{aligned}$$

This observes that a state of $P \setminus X$ is only stable when the corresponding state of P is not only stable but also refuses X.

$$\begin{aligned}
divergences(P \setminus X) = {} & \{(s \setminus X) \hat{} t \mid s \in divergences(P)\} \\
& \cup \{(u \setminus X) \hat{} t \mid u \in \overline{traces_\perp(P)} \wedge (u \setminus X) \text{ is finite}\}.
\end{aligned}$$

Here, $\overline{X}$, for $X \subseteq \Sigma^*$, is the set of all infinite traces all of whose finite prefixes are in X. This definition relies on P being finitely nondeterministic, since this brings König's Lemma into play, allowing us to infer infinite traces from the set of finite ones.

$$\begin{aligned}
failures_\perp(P[\![R]\!]) = {} & \{(s', X) \mid \exists s.s\, R\, s' \wedge (s, R^{-1}(X)) \in failures_\perp(P)\} \\
& \cup \{(s, X) \mid s \in divergences(P[\![R]\!])\}.
\end{aligned}$$

Here, $R^{-1}(X) = \{a \mid \exists a' \in X.(a, a') \in X\}$ is the set of all events that map to X under R. Note that we also use R extended to traces.

$$divergences(P[\![R]\!]) = \{s' \hat{} t \mid \exists s \in divergences(P) \cap \Sigma^*.s\, R\, s'\}.$$

Every CSP term represents a monotone[8] operator over $\mathcal{N}$ with respect to $\sqsubseteq_{FD}$. It follows by Tarski's theorem that each CSP-defined operator $F(P)$ from $\mathcal{N}$ to itself has a $\sqsubseteq_{FD}$-least fixed point. This is the denotation of the recursive term $\mu p.F(p)$. This fixed point is the operationally correct one for $\mathcal{N}$. Different fixed-point theories are required for other CSP models.

[8] Indeed, every finitely nondeterministic term represents a continuous operator, meaning that it preserves the least upper bounds of linearly ordered sets under $\sqsubseteq_{FD}$.

32.2.2.4 Timed CSP

Though we do not go into detail here, we remark that a real-time version of CSP called Timed CSP was introduced by Reed and Roscoe [70] and extensively developed in works such as [81]. This takes essentially the same operators as the untimed CSP we have seen to date and gives them an exact real-time interpretation, usually with the non-negative real numbers as the times at which events occur. The only extra operator that is necessary is the process $WAIT(t)$, which waits for t time units before terminating with $\checkmark$. (Other timing constructs such as time-out can be defined in terms of this and the other CSP operators.)

The same basic philosophy of defining equivalence through behaviorally based models was used for Timed CSP. However, despite the similarity of the languages, the considerations of time mean that the models for Timed CSP look rather different from those we have seen already. For example, divergence plays a much more minor role, and in order to get a compositional model it is necessary to record what actions a process refuses at every instant during a behavior.

The most usual fixed-point theory for the denotational semantics of recursion in Timed CSP uses the contraction mapping theorem for complete metric spaces, rather than Tarski's theorem as above.

32.2.3 ACP: Process Algebra via Equational Semantics

In this section, we present the process algebra ACP [9] and its equational theory. ACP also has an LTS-style semantic theory in the form of a *graph model* and corresponding notion of bisimulation. We omit this from our discussion of ACP, in order to focus on its equational theory, which is ACP's hallmark.

As is the practice in ACP, we begin with the theory BPA (Basic Process Algebra), which describes processes constructed from constants, plus, and sequential composition. We will then add to BPA a notion of parallel composition (merge and left-merge) to obtain PA (Process Algebra. Finally, the theory ACP is derived by extending BPA with the constant δ (for deadlock), a combined notion of parallel composition and communication, and a restriction operator.

32.2.3.1 BPA

The signature $\Sigma(\mathrm{BPA}(A))$ consists of one sort P (for processes) and three types of operators: constant processes a, for each atomic action a, the sequential composition (or sequencing) operator '$\cdot$', and the alternative composition (or nondeterministic choice) operator '$+$'. The set of all constants is denoted by A, and is considered a parameter to the theory.

$$\Sigma(\mathrm{BPA}(A)) = \{a : \to \mathsf{P} | a \in A\} \cup \{+ : \mathsf{P} \times \mathsf{P} \to \mathsf{P}\} \cup \{\cdot : \mathsf{P} \times \mathsf{P} \to \mathsf{P}\}.$$

The axiom system BPA(A) is given by:

$x + y = y + x$	A1
$(x + y) + z = x + (y + z)$	A2
$x + x = x$	A3
$(x + y) \cdot z = x \cdot z + y \cdot z$	A4
$(x \cdot y) \cdot z = x \cdot (y \cdot z)$	A5

Note the absence of the axiom $x \cdot (y + z) = x \cdot y + x \cdot z$, which does not hold in ACP's bisimulation model.

In the setting of BPA(A), bisimulation, denoted $\leftrightarrows$, is a congruence (see, e.g., [7]).

Proposition 1 *If* $G_1 \leftrightarrows G_2$, *then* $G + G_1 \leftrightarrows G + G_2$, $G \cdot G_1 \leftrightarrows G \cdot G_2$, *and* $G_1 \cdot G \leftrightarrows G_2 \cdot G$.

We have that $\mathscr{A}\mathscr{P}\mathscr{G}(A, N)/\leftrightarrows$, the *graph model*, is indeed a model of the axiom system BPA(A), and that BPA(A) constitutes a complete axiomatization of process equivalence in $\mathscr{A}\mathscr{P}\mathscr{G}(A, N)/\leftrightarrows$.

Theorem 2 ([7])

1. $\mathscr{A}\mathscr{P}\mathscr{G}(A, N)/\leftrightarrows \models \mathrm{BPA}(A)$
2. *For all closed expressions* p, q *over* $\Sigma(\mathrm{BPA}(A))$:

$$\mathscr{A}\mathscr{P}\mathscr{G}(A, N)/\leftrightarrows \models p = q \implies \mathrm{BPA}(A) \vdash p = q.$$

32.2.3.2 PA

The signature $\Sigma(\mathrm{PA}(A))$ is obtained from $\Sigma(\mathrm{BPA}(A))$ by adding an interleaving *merge* operator $\|$ and a *left-merge* operator $\lfloor\!\lfloor$.

$$\Sigma(\mathrm{PA}(A)) = \Sigma(\mathrm{BPA}(A)) \cup \{\|\colon \mathsf{P} \times \mathsf{P} \to \mathsf{P}\} \cup \{\lfloor\!\lfloor\colon \mathsf{P} \times \mathsf{P} \to \mathsf{P}\}$$

Intuitively, the process $x \| y$ is obtained by interleaving (shuffling) the atomic actions of x and y together. Left-merge is an auxiliary operator in that it permits $\|$ to be specified in finitely many equations. The process $x \lfloor\!\lfloor y$ has the same meaning as $x \| y$, but with the restriction that the first step must come from x.

The axiom system PA(A) is given by:

BPA(A)+

$x \| y = x \lfloor\!\lfloor y + y \lfloor\!\lfloor x$	M1
$a \lfloor\!\lfloor x = a \cdot x$	M2
$(a \cdot x) \lfloor\!\lfloor y = a \cdot (x \| y)$	M3
$(x + y) \lfloor\!\lfloor z = x \lfloor\!\lfloor z + y \lfloor\!\lfloor z$	M4

Again one may notice that $\leftrightarrows$ is a congruence, $\mathscr{A}\mathscr{P}\mathscr{G}(A, N)/\leftrightarrows \models \mathrm{PA}(A)$ and (see [7]) PA(A) constitutes a complete axiomatization of process equivalence in $\mathscr{A}\mathscr{P}\mathscr{G}(A, N)/\leftrightarrows$.

32.2.3.3 ACP

The equational system ACP treats the operators of BPA(A) as well as the new constant δ representing deadlock; a *communication merge* operator $|$ describing the result of a communication between any two atomic actions; a *merge* operator $\|$ and *left-merge* operator $\lfloor\!\lfloor$ like those of PA(A) but which additionally admit the possibility of communication; and a family of restriction operators ∂_H, $H \subseteq A$. We will also need an auxiliary operator I that defines the initial actions that a process can perform.

Letting $A_\delta = A \cup \{\delta\}$, the signature of $\mathrm{ACP}_I^-(A)$ extends that of PA(A) as follows:

$$\Sigma(\mathrm{ACP}_I^-(A)) = \Sigma(\mathrm{PA}(A)) \cup \{\delta :\to \mathsf{P}\} \cup \{| : \mathsf{P} \times \mathsf{P} \to \mathsf{P}\}$$
$$\cup \{\partial_H : \mathsf{P} \to \mathsf{P} \mid H \subseteq A\} \cup \{I : \mathsf{P} \to 2^{A_\delta}\}$$

It is convenient to define communication merge as the extension of a binary commutative and associative function on atomic actions (i.e., $| : A_\delta \times A_\delta \to A_\delta$) with δ acting as a multiplicative zero. This is accomplished with axioms C1–3 below. We further require $|$, restricted to $A_\delta \times A_\delta$, to be total and this is expressed by the following axiom:

$$\boxed{\forall a, b \in \mathsf{P}.\ \overline{A_\delta}(a) \wedge \overline{A_\delta}(b) \implies \overline{A_\delta}(a|b) \quad \text{C0}}$$

Here, $\overline{A_\delta}$ is the characteristic predicate of A_δ:

$$\overline{A_\delta}(x) = \bigvee_{a \in A_\delta} (x = a).$$

The axioms of $\mathrm{ACP}_I^-(A)$ are now given. In this system, a, b, c range over A_δ, $H_\delta = H \cup \{\delta\}$, for $H \subseteq A$, and $\cap$, $\cup$ are used on 2^{A_δ} without further specification.

BPA(A)+

$x + \delta = x$	A6
$\delta \cdot x = \delta$	A7

+

C0+

$a \mid b = b \mid a$	C1
$(a \mid b) \mid c = a \mid (b \mid c)$	C2
$\delta \mid a = \delta$	C3

+

$x \parallel y = x \between y + y \between x + x \mid y$	CM1
$a \between x = a \cdot x$	CM2
$(a \cdot x) \between y = a \cdot (x \parallel y)$	CM3
$(x + y) \between z = (x \between z) + (y \between z)$	CM4
$a \mid (b \cdot x) = (a \mid b) \cdot x$	CM5
$(a \cdot x) \mid b = (a \mid b) \cdot x$	CM6
$(a \cdot x) \mid (b \cdot y) = (a \mid b) \cdot (x \parallel y)$	CM7
$(x + y) \mid z = x \mid z + y \mid z$	CM8
$x \mid (y + z) = x \mid y + x \mid z$	CM9

+

$I(a) = \{a\}$	I1
$I(x \cdot y) = I(x)$	I2
$I(x + y) = I(x) \cup I(y)$	I3

+

$a \in H \implies \partial_H(a) = \delta$	D1
$a \notin H \implies \partial_H(a) = a$	D2
$\partial_H(x + y) = \partial_H(x) + \partial_H(y)$	D3
$\partial_H(x \cdot y) = \partial_H(x) \cdot \partial_H(y)$	D4

As noted on the Wikipedia page for ACP and as can be seen in our treatment of ACP in this chapter, the development of ACP focused on the *algebra* of processes (more so than CCS and CSP), and sought to create an abstract, generalized axiomatic system for processes [57]. In fact, the term *process algebra* was coined during the research that led to ACP.

32.3 Algorithms and Methodologies

32.3.1 Bisimulation and Simulation

In the case of finite-state CCS processes, that is, those whose underlying LTSs are finite-state, we have the following main result. Let p and q be finite-state processes whose underlying LTSs have a total of n states and m transitions. Then, as was shown by Kanellakis and Smolka [47], whether or not p and q are bisimilar can be decided in polynomial time, $O(nm)$ time to be exact. This algorithm, which has come to be known as *Relational Coarsest Partitioning* (RCP), was subsequently improved upon by Paige and Tarjan who devised one that runs in $O(m \log n)$ time [63]. This is in stark contrast to the equivalence problem for regular expressions, which was shown to be PSPACE-complete [45].

The Kanellakis–Smolka algorithm exploits the fact that an equivalence relation on a set of states may be viewed as a *partition*, or set of pairwise-disjoint subsets (called *blocks*) of the state set whose union is the state set. In this representation,

```
function split(B, a, B') =
  {{s ∈ B | ∃s' ∈ B'. s -a-> s'}, {s ∈ B | ¬∃s' ∈ B'. s -a-> s'}} − {∅};

algorithm bisim(Q, A_CCS, ⟶);
  begin
    P1 := {Q};
    P2 := ∅;
    while P1 ≠ P2 do begin
      P2 := P1;
      P1 := ∅;
      foreach B ∈ P2 do P1 := P1 ∪ split(B, a, B');
    end
  end
```

Fig. 2 The Relational Coarsest Partitioning algorithm for bisimulation equivalence

blocks correspond to the equivalence classes—so two states are equivalent exactly when they belong to the same block. Beginning with the partition containing one block (representing the trivial equivalence relation consisting of one equivalence class), the algorithm repeatedly *refines* this partition (by splitting blocks) until the associated equivalence relation becomes a bisimulation.

In order to determine whether the partition needs further refining, the algorithm looks at each block in turn. If a state in a block B has an a-derivative in a block B' and another state in B does not, then the algorithm splits B into two blocks, one containing the states having an a-derivative in B' and the other containing the states that do not. When no more splitting is possible, the resulting equivalence relation corresponds exactly to bisimulation equivalence on the given transition system.

The algorithm is given in Fig. 2. It takes as input the (finite-state) LTS $(Q, A_{CCS}, \longrightarrow)$, and computes as output the partition P_1 of Q corresponding to bisimulation equivalence. Function `split` is used to split one block with respect to another; notice that $\texttt{split}(B, a, B') = \{B\}$ (i.e., B is not split with respect to a and B') if either all the states in B, or none of them, have an a-derivative in B'. It should also be pointed out that $P_1 = P_2$ exactly when no more splits in P_1 are possible. The worst-case complexity of *bisim* is $O(|\longrightarrow| * |Q|)$.

Bisimulation was originally defined by Milner as the limit of a sequence of successively finer equivalence relations, $\sim_k$, where $\sim_1$ is trace equivalence. Kanellakis and Smolka also showed that, for each fixed k, deciding $\sim_k$ is PSPACE-complete, a complexity that disappears in the limit; i.e., upon reaching $\sim$.

As for weak bisimilarity on finite-state processes, one can first pre-compute the weak transition relation (which simply amounts to computing the transitive closure) and construct new regular processes where transitions are replaced with weak transitions. On these new regular systems the algorithms for (strong) bisimilarity checking can be used. Hence the problem for weak bisimilarity can also be decided in polynomial time.

There are also algorithms for computing similarity relations of both finite-state and infinite-state processes. In [38], an $O(mn)$ algorithm is presented for computing the similarity relation of a finite-state process with n states and m transitions.

For effectively presented infinite-state processes, they present a symbolic similarity-checking procedure that terminates if a finite similarity relation exists.

32.3.2 Checking Refinement over Behavioral Models

A few years after the discovery of trace equivalence [41] and $\mathscr{N}$, Kanellakis and Smolka [47] showed that equivalence checking based on this type of model is potentially very expensive: PSPACE-complete in the size of the state spaces of the processes involved (itself frequently exponential in the number of their parallel components). There was therefore a period of several years during which model checking based on such models seemed unattractive compared with the polynomial (in state space) algorithms which exist to check bisimulation-style equivalences. After all, equivalence checking and refinement checking both reduce easily to one another, when we note that $P \sqsubseteq Q \Leftrightarrow P \equiv Q$ over such models. However, as we will see shortly, things were not as bad as they seemed.

FDR (standing for Failures Divergences Refinement) is a refinement checker for CSP that was first released in 1991 after its initial development as part of a hardware verification project. It is presently maintained by the Oxford University Department of Computer Science. Its primary functionality is to check refinement in a number of CSP's behavioral models, including those discussed in Sect. 32.2.2. The capabilities and limitations of FDR are much more fully set out than we have space for here in Roscoe's 2010 book [75] and more recent papers [3, 4, 32].

In deciding whether $Spec \sqsubseteq Imp$, it first determinizes, or *normalizes*, $Spec$. In other words, it transforms $Spec$ into a form that has a unique state for every trace. It is in this transformation that FDR's algorithm has the potential to exhibit the exponential complexity we would expect when solving a PSPACE-complete problem. For in calculating the normal form of $Spec$, it is natural to discover the set of *subsets* of $Spec$'s states that have some trace in common: in pathological cases, the number of such subsets can be exponential.

Such pathology—or indeed anything close to it—is uncommon in cases where $Spec$ is a coherently designed process, and almost unknown in cases where $Spec$ is a process representing a clear behavioral specification. In the great majority of FDR use cases the process on the left-hand side of refinement is a specification with relatively few states, and the time taken to normalize it is trivial.

Given a normalized specification $Spec$ and an implementation Imp, FDR seeks to prove that there is no state reachable in $Spec \times Imp$, where Imp displays a behavior (typically an event, refusal set, or divergence) that $Spec$ does not. Either it covers all reachable states without finding such a situation, or it finds a counter-example and displays the way Imp performs it. This is, of course, just model checking. If $Spec \times Imp$ were explored one state at a time, it follows that the complexity of this phase is at worst the product of the (normalized) state space of $Spec$ and the (unnormalized) one of Imp. In most cases, just one $Spec$ normal-form state is visited per state of Imp.

32.3.3 Diagnostic Information (HML, $\sqsubseteq_{FD}$)

In [15], Cleaveland presented an algorithm for generating an HML formula that differentiates between two bisimulation-inequivalent finite-state systems. The method works in conjunction with the Relational Coarsest Partitioning algorithm for computing bisimulation equivalence (see Sect. 32.3.1) and yields formulae that are often minimal in a precisely defined sense.

The method of [15] uses information computed by a slightly altered version of the RCP algorithm that, in addition to computing the partition as described in Sect. 32.3.1, retains information about how blocks are split. Then, a postprocessing step constructs a formula distinguishing the states in one block from the states in another.

The RCP algorithm is modified as follows. Rather than discarding an old partition after it is refined, the new procedure constructs a "tree" of blocks as follows. The children of a block are the new blocks that result when the block is split; accordingly, the root is labeled with the block Q, and after each iteration of the `foreach` loop the leaves of this tree represent the current partition. When a block B is split (by $split(B, a, B')$), we place the new block $B_1 = \{ s \in B \mid \exists s' \in B'.\ s \xrightarrow{a} s' \}$ as the left child and the new block $B_2 = \{ s \in B \mid \neg\exists s' \in B'.\ s \xrightarrow{a} s' \}$ as the right child, and we label the arc connecting B to B_1 with a and B'. Recall that every state in B_1 has an a-transition into B' and that no state in B_2 does. If a block is not split during an iteration of the `foreach` loop, it is assigned a copy of itself as its only child.[9]

Given a block tree computed by the new version of the RCP algorithm, and two states s_1 and s_2 that are inequivalent and hence in different blocks, the postprocessing step builds a formula $\delta(s_1, s_2)$ that distinguishes $\{s_1\}$ from $\{s_2\}$. Although this formula will not necessarily be minimal, it will in general be much smaller than the formula computed using the more naive method described in [15]; it is guaranteed to be no larger.

The more naive method computes distinguishing formulae by associating a formula, $\Phi(B)$, with each block B in the partition computed by RCP in such a way that the following hold.

- $B \subseteq [\![\Phi(B)]\!]$.
- $B' \cap [\![\Phi(B)]\!] = \emptyset$ if $B' \neq B$.

In the initial partition, $\{Q\}$, $\Phi(Q)$ is set to tt. Now suppose a block B is split, i.e., suppose there is an action a and another block B' such that $split(B, a, B') = \{B_1, B_2\}$, with every state in B_1 having a transition into B' and no state in B_2 having one. Then $\Phi(B_1)$ may be set to $\Phi(B) \wedge \langle a \rangle \Phi(B')$, while $\Phi(B_2)$ becomes $\Phi(B) \wedge \neg\langle a \rangle \Phi(B')$. Arguing inductively, it is easy to establish that for any block B, a state satisfies $\Phi(B)$ exactly when it is contained in B. Since two states that are not bisimulation equivalent will eventually wind up in different blocks, it is a simple

[9] Strictly speaking, this is not necessary; these blocks may be left childless. These spurious children are included to simplify the inductive argument of correctness.

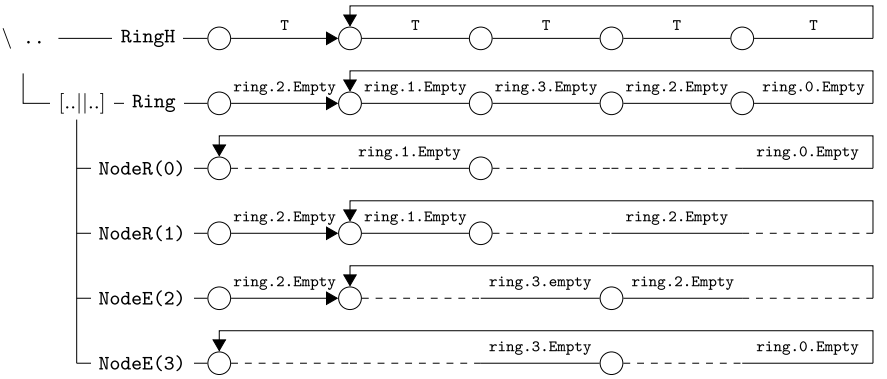


Fig. 3 Debugging a divergence in FDR3

matter to compute a formula that distinguishes such states: just return the formula associated with one of the containing blocks.

The diagnostic offered for failure of a refinement $P \sqsubseteq Q$ over a behavioral model is invariably a behavior b of Q that is not possible for P. We can generally demonstrate reasonably succinctly why such a behavior is possible. This is of course appropriate for the case where P represents some sort of specification process. It is not really possible, where this is desired, to give an explanation for why P *does not* permit such a behavior, were we to decide that b is OK and therefore P is at fault for not permitting it: a behavior might have been allowed in many different ways. We therefore concentrate on the analysis of Q's offending behavior b, which can be done by providing an explanation of how each component process in Q contributes to b, perhaps coupled with animation.

The FDR debugger permits one to see the contribution (which, for familiar models, will always be a trace, a failure, or a divergence) of each component process to a counter-example. Thus in $(P \underset{X}{\parallel} Q) \setminus X$ the user can look at the top-level behavior, that of $P \underset{X}{\parallel} Q$ (so with all the hidden events revealed), that of P, and that of Q. The lowest-level components one can examine are those identified for direct translation to state machines by the compiler.

Like previous versions of FDR, FDR3 permits one to examine the behaviors of both the top-level system and its components. However, unlike its predecessors, FDR3 allows users to inspect all these component behaviors at the same time and see how they relate to each other, and at the same time allows one to animate the behavior of the whole of P or some component thereof. For example Fig. 3 illustrates a divergence in a small token ring: two nameless[10] tokens (initially in nodes 0 and 1—they are initially in state `NodeR` rather than the empty `NodeE`—simply circulate

[10]Careful examination of the behavior shows that the loop that FDR has detected actually swaps the position of the two tokens.

when the nodes do not want to use them, creating an infinite sequence of τs. In this view we have chosen to view all the available details: note that top-level actions typically map down to actions in a subset of the low-level components, with dotted lines meaning that a given component is not participating in the corresponding event.

The figure illustrates the most general form of a divergence error: an initial sequence of τ and non-τ events followed by a loop of events that are all, in the top-level view, τs. Trace and refusal errors simply produce a sequence of states, with the addition of final refusal (alternatively acceptance) sets for each process.

The user is able to inspect each state of every component by mouse clicks on the window, or to animate them.

This extended functionality is non-trivial in the case where some of the components are the subject of state compression functions, particularly because these frequently make τs that occur inside the compression disappear when viewed from the outside. The explanation that FDR3 produces for the counter-example removes this effect of compression, so such τs are made explicit in the top level behavior even though they may not have been seen in the main refinement checking phase. Aligning such "decompressed" behaviors is challenging for divergent loops.

FDR3 (like FDR2) allows one to control how many counter-examples are reported for a single check, the default being one: if this is set to a higher value the tool will carry on until it has found the specified number of states exhibiting a breach of the specification or the state space is exhausted. As reported in Chap. 15 of [75], this capability can be exploited to return quantitative information, such as a time bound, about the process under examination.

FDR3 provides a tool which visualizes the transitions of any process and allows the user to inspect any state in more detail.

32.3.4 Compositional Verification

All process algebras have well-developed theories of process equivalence $\equiv$, as outlined in Sect. 32.2, and some such as have well-developed theories of refinement in which $P \sqsubseteq Q$ means in some sense that Q is a refinement of P and so can replace it without losing correctness. Both these ideas, except for a few exceptional cases, are always developed in a fully compositional way, in the senses that

- If $P \equiv Q$, then $C[P] \equiv C[Q]$ for any context $C[\cdot]$ formed in the given process algebra.
- If $P \sqsubseteq Q$, then $C[P] \sqsubseteq C[Q]$ for such contexts.

The first of these underpins the use of compression operators by FDR and other model checkers: if we take a complex process $\mathbf{P} = C[P_1, \ldots, P_n]$ and can find processes Q_i such that $P_i \equiv Q_i$ but Q_i has fewer states than P_i, then if we want to check $\mathbf{P}$ for some property R that respects $\equiv$ (i.e., $P \equiv Q \Rightarrow (R(P) \Leftrightarrow R(Q))$) then it is usually easier to check $\mathbf{Q} = C[Q_1, \ldots, Q_n]$, since the latter can be expected to have fewer states.

Such manipulations can also be done by hand: if, for such a property R we can find a simpler Q such that $P \equiv Q$, then to prove $C[P]$ satisfies R it is sufficient to prove $R(C[Q])$.

32.4 Tools

32.4.1 FDR

FDR has gone through three major releases, FDR (1991), FDR2 (1995), and FDR3 (2013) [32], with the last two representing complete re-writes and incorporating significant algorithm changes. FDR3 can be downloaded from www.cs.ox.ac.uk/projects/fdr. FDR2 had many incremental releases up to FDR2.94 [3]. FDR2 introduced the CSP_M language, which merges the CSP process algebra with a lazy functional language in the style of Haskell. This combination, introduced by Scattergood [80], allows functional programming to be used for laying out networks, for defining sets of events such as process alphabets, and for computations involving the events and parameters used by individual processes. It makes CSP into a serious programming language that can provide succinct descriptions of substantial systems.

FDR2 and FDR3 use essentially the same method of enumerating and exploring state machines. This is optimized to deal with the case of networks formed as the parallel composition of relatively small components. One of the main functions of the CSP_M compiler is to identify these low-level components; it then compiles each of them into an explicit state machine and derives rules (supercombinators) for combining actions of these components into actions of the whole system. It therefore ends up with an efficient implicit representation of the state space of the implementation that can either be explored explicitly by bit-vector operations or translated into propositional logic for SAT checking. An implementation of the latter was reported in [64], and incorporated into later experimental releases of FDR2 [3], together with a CEGAR implementation and the Static Livelock analysis (SLAP) techniques reported in [62]. Explicit model checking, however, supplemented by compression (i.e., state-space reduction) techniques such as strong and divergence-respecting weak bisimulation, diamond compression, and normalization, and by the *chase* operator, generally remains the most effective approach.

The main search mode of FDR has been breadth-first (BFS) because this produces minimal-length counter-examples and is comparatively efficient in its use of virtual memory and file storage. Implemented using B-Trees, it works particularly well with the backing store provided by Solid-State Drives (SSDs).[11] DFS is used to

[11]For example, FDR3 performed a 73-billion-state check, using 707 G of storage on a 16-core machine with 256 G of RAM and a RAID array of 8 SSDs. The rate of state coverage remained at just over 700,000 per second over almost the entire check, taking about 29 hours in all. Thus, there was no slow-down at the point where the disk storage started to be used, or beyond it for reasons attributable to storage access speed.

identify divergence, and can be useful for quickly finding counter-examples when a system has many.

The most significant innovations in FDR3 are:

- Many functions are parallelized for multi-core and clusters. This and other efficiency improvements reduce FDR's most familiar benchmark example, namely evaluating all 187M states of the standard peg solitaire board (see [72]) to perhaps 5–10 minutes on a standard workstation or laptop with 2–6 cores, 51 seconds on a 40-core server and less that 2 seconds on a supercomputing cluster of 64 16-core machines. Over a wide range of examples, the current parallel implementation of this main model-checking phase works well in terms of processor usage and obtaining near-linear speed-up. The multi-core work is reported in [32] and the cluster implementation in [33]. Experimental results reported in these papers suggest slightly super-linear speed-up on average. The latter paper reports the completion of a check with 1.2×10^{12} explicit states on a cluster of 64 16-core machines rented from Amazon. This took about 5 hours, and would have had about 10^{20} states if FDR's compressions had not been used.
- FDR3 has the potential to support languages other than CSP, provided that their semantics is CSP-like as discussed above. Future versions of FDR3 will provide support for users wishing to define their own input language subject to this constraint.
- FDR3 has an integrated type checker for the CSP_M language.
- The debugger (i.e., counter-example viewer) has been improved.

There have been several experiments in the context of CSP with using symbolic model-checking techniques such as BDDs [87] and the SAT checking mentioned above. While these have been successful in finding counter-examples in some classes of system (some reasonably up-to-date figures for the SAT case may be found in [64]; ARC is not maintained, and we were unable to obtain a version to compare), they have not so far, at least to our knowledge, performed significantly better than FDR with DFS, and have performed relatively badly [64] (even with the incompleteness of bounded model checking) for cases where there is no counter-example.

FDR has been extended [4] to encompass the theory of Timed CSP, using Ouaknine's [61, 75] translation to a discrete-time language, where continuous properties can be inferred using the theory of digitization (first proposed for timed automata in [39]). The analysis of timed systems requires the use of priority: internal τ actions must have priority over the passage of time. Besides Timed CSP, FDR3 and later versions of FDR2 support the priority operator described in Chap. 20 of [75]. Though this operator is only compositional over the finest abstract models of CSP, it adds significantly to the language's expressive power; see [76, 77].

32.4.1.1 Using FDR

FDR is largely written in C++, and is currently only supported on Unix platforms (including Mac OSX), though there are plans for a Windows port of FDR3.

There are three ways of using FDR. Traditionally, it was used either through its own GUI or through a command-line interface. In addition, FDR3 has an API which is starting to replace the command-line interface when integrating it as the back-end for other tools such as SVA (see below).

The following are some of the major applications of FDR.

- It was the first general-purpose model checker applied to cryptographic protocols, by Gavin Lowe [54], who for over a decade has supported a front-end (Casper [55]) for FDR that is dedicated to this application.
- The CSP_M language has been used to write "compilers" for other notations, such as a shared-variable language in SVA (see Chaps. 18 and 19 of [75]), and Statemate Statecharts in [78]. These compilers actually work by simulating the object program in CSP; the results of the simulation are then analyzed by FDR.
- FDR is integrated into ASD, a technology for developing verified embedded control software [44]. (As in the examples above, the end-user does not see the CSP that is generated.) Hundreds of millions of lines of verified code, for systems typically in the 10s or 100s of thousands of lines of code, have been generated using ASD for some substantial clients.
- It has been used on many substantial projects, both in research and product development, by DERA/QinetiQ; see [89], for example. These include aspects of the European Fighter Aircraft (Typhoon) and military networking.
- It has been used by over 30 commercial and government organizations in total, mainly in areas relating to safety-critical systems but also in more diverse application development [49] and test-suite generation [65]. Many of these applications have been based on real-time systems, using the discrete-time variant of CSP described in Chap. 14 of [72]. This is distinct from the support for Timed CSP discussed earlier, though it too requires the prioritization of internal events over the passage of time.

FDR is the work of many people, including Michael Goldsmith, David Jackson, Paul Gardiner, Bryan Scattergood, and Philip Armstrong. Tom Gibson-Robinson, aided by Sasha Boulgakov and Armstrong, has led the development of FDR3. The project has been led throughout by Roscoe and supported by funding from EPSRC, ONR, DARPA, DERA/QinetiQ, and industrial users. For the period 1991–2007, it was developed and maintained by Formal Systems (Europe) Ltd., a small Oxford University spin-off, and since 2008 by Roscoe's group at Oxford University, Department of Computer Science.

32.4.2 The Concurrency Workbench

The Concurrency Workbench (see http://www.cs.sunysb.edu/~cwb and [19–22]) is an extensible tool for verifying systems written in various process algebras, including CCS. The key feature of the system is its modular design and concomitant flexibility. The system is built around three generic algorithms: one for computing

equivalences (the general equivalence is based on *bisimulation equivalence*), one for computing preorders (the general preorder is based on the *divergence preorder* of [84]), and one for *model checking* in a very rich temporal logic, the propositional mu-calculus [48]. The system uses the first two of these generic routines to compute a number of different equivalences and preorders, including the failures/testing relations, by combining them with suitable *process transformation routines* [16, 17]. To decide a given relation, the Workbench applies the appropriate transformation to the processes in question and then runs the implied generic routine on the transformed processes. This structure also makes it particularly easy to add new process relations to the Workbench—just determine the appropriate process transformation and apply the indicated general procedure.

The *model-checking* facility of the Workbench enables users to determine when a process satisfies a formula written in the propositional mu-calculus. The appeal of the mu-calculus lies in its expressive power and its ability to encode many other temporal logics (and, indeed, equivalences and preorders [83]) in a uniform fashion [14, 30, 83]. This power results from its capacity for expressing arbitrary *recursive* formulae. The Workbench also supplies a *macro* facility that enables new propositional constructors to be defined in terms of existing ones. This feature enables the development of other temporal logics in the Workbench: one defines the proposition constructors of the logic as macros, and the model checker may then be used to check formulae built using these macros.

The inclusion of these different verification techniques permits different styles of correctness checking to be carried out, and it facilitates the development of methodologies that employ more than one of these techniques.

32.4.3 XMC

XMC (see http://www.cs.sunysb.edu/~lmc/ and [69]) supports the specification, simulation, and verification of concurrent systems such as communication protocols and embedded systems. It is implemented atop XSB, a high-performance logic-programming system. System models are specified in XL, a typed value-passing language based on Milner's CCS, properties of interest are specified in the modal mu-calculus, and model checking is used to verify properties of systems. XMC incorporates a *justifier* which allows the user to navigate the proof tree underlying a model-checking computation; such proof trees are effective in debugging branching-time formulae. XMC has been successfully applied to the specification and verification of a variety of systems including the Rether real-time Ethernet protocol [28], the Java meta-locking algorithm [8], and the SET e-commerce protocol [56].

32.4.4 mCRL2

mCRL2 [35] is a toolset based on an extended version of ACP. Extensions at the process-algebra level include the addition of renaming and hiding in forms similar

to those used in CSP, plus support for time. The language is extended with data in a form not dissimilar to the extension from CSP to CSP_M. As with XMC above, properties of interest are specified in the modal mu-calculus. It operates by reducing processes to LTSs and uses state minimizations such as strong and branching bisimulation [34]. It can act as a front-end to LTS analysis tools such as LTSmin [11], and is capable of dealing with systems with billions of explicit states.

mCRL2 has been widely used in industrial verifications in areas such as bridge control, and in the development of software for CERN [46], the ALMA radio telescope [66], and a solar-powered car. Some of these are described at mCRL2's web page: http://www.mcrl2.org.

32.5 Case Studies

32.5.1 Distributed Leadership Election (FDR)

Many distributed algorithms require the presence of a leader, or coordinator process, which plays some central role such as initiating phases or collecting results. *Leadership election* algorithms are designed to secure agreement throughout the network about the identity of such a process. The details of the problem vary with the network in question: is the topology fixed (e.g., a ring), arbitrary but static, or dynamic? What initialization do the nodes have (e.g., with different integers)? Do we have to cope with node or network errors, and/or nodes dying and reviving? Can the leadership role remain static, or is it expected to be shared among the nodes?

Many election algorithms are based on asynchronous communications; it is, however, straightforward to simulate this in a process algebra with handshake communication such as CCS or CSP. Algorithms described for highly symmetric networks are sometimes stochastic, as when all processes choose a value at random to determine which is the leader, and it is possible that some may pick the same value. Most depend on timing ideas such as time-outs. CSP and FDR are capable of handling the first and third of these ideas, but not the analysis of randomized algorithms. A good example to examine, therefore is the *bully* algorithm. This algorithm was first described by Garcia-Molina [31]. A full case study is set out in Chap. 14 of [75]. The CSP_M code of that example (as well as much other such code) can be downloaded from that book's website http://www.cs.ox.ac.uk/ucs.

The following paraphrases the presentation of the bully algorithm in [23].

> This is used to allow a group of N processors to have an agreed *coordinator* from among them. It is intended to work despite processors failing and reviving from time to time.[12] The objective is always to have them agree that the coordinator is the highest-indexed processor that is working. We assume that communication is reliable, but processors can fail at any time (including during the election procedure that the algorithm represents).

[12]The algorithm is therefore intended to be fault-tolerant.

> There are three types of message in an election, and in our implementation these are represented by separate channels rather than as data values sent along a single channel of more complex type. A process sends an `election` message to all those with a higher index. It then awaits an `answer` message in response. If none arrives within time `T1`, the processor considers itself the coordinator (as it thinks all the higher ones must be dead), and sends a `coordinator` message to all processes with lower identifiers announcing this fact. Otherwise, the process waits a further limited period (`T2`), expecting to receive a coordinator message as described in the next paragraph, before beginning another election (if no such message is received).
> If a process receives a coordinator message, it records the identifier of the coordinator contained within it (i.e., the sender of that message) and treats that process as the coordinator.
> If a process receives an `election` message, it sends back an `answer` message and begins another election, unless it has begun one already.
> When a failed process is `revived`, it begins an election. If it has the highest live process identifier, then it will eventually decide that it is the coordinator, and announce this to the other processes. Thus it will become the coordinator, even though the current coordinator is functioning. It is for this reason that the algorithm is called the *bully* algorithm.

The following are the channels[13] that the algorithm description refers to explicitly:

```
channel election, answer, coordinator:Proc.Proc
```

where `Proc={0..N-1}` and all these represent messages from a sender process (the first index) to a receiver.

An `election.n.m` message is used to announce an election (from `n` to `m`); an `answer` message is sent in response to an election message; and a `coordinator` message is sent to announce the identity of the new coordinator. A process begins an election when it notices that the coordinator has failed.

When you try to translate this into a language like CSP you realize that a lot of detail has been left out. How do processes notice that the coordinator has failed? How regularly are such failures tested for? It seems inappropriate to implement the bully algorithm in such a way that any process (even a failed one) can block another one from outputting to it. So what happens if processes receive messages that they are not currently expecting? Developing the example using FDR rapidly makes you allow for many such cases, and it is not always easy to decide what the right reaction is to such messages. A typical node state therefore has code to deal with many sorts of input. For example:

```
SendElections(k,n) =
(if k<=n then AwaitAnswers(T1,n)
         else election.n.k -> SendElections(k-1,n))
[] election?k':below(n)!n ->answer.n.k' ->
                            SendElections(k,n)
[] answer?k:above(n)!n -> AwaitCoordinator(T2,n)
[] ok?k':above(n)!n -> SendElections(k,n)
```

[13] The key word `channel` declares one or more channels in CSP_M that have the type set out after the colon.

```
[] coordinator?c:above(n)!n -> Running'(n,c)
[] fail.n -> Failed(n)
[] test?m:below(n)!n  ->  SendElections(k,n)
```

where `test` and `ok` are channels used to test coordinators and respond to tests. One event that the above state cannot communicate initially is `tock`, the event representing time passing. This is because the above state is *urgent*: in the absence of interruption by other events that happen immediately, this state always sends the message `election.n.k` on the current time step. The node can also be in states `AwaitAnswers`, `AwaitCoordinator`, `BecomeCoordinator`, `RunAsCoord`, `Running` `Testing`, and `Failed`.

Discovering how each of these states should react to any of the messages that might arise without some obvious misbehavior took a number of iterations consisting of checking small implementations of the algorithm against both general-purpose specifications such as the consistency of the timing model, and case-specific ones – for example, that if a test has occurred subsequent to any node failure, sufficiently long before the present, then all nodes have the correct view of who the coordinator is.

Eventually a point was reached where the system was provably correct for three nodes. Trying it for four, however, resulted in many counter-example traces like the following:

```
<fail.2, fail.3, test.1.3, tock, election.1.3,
 election.1.2, revive.2, revive.3, coordinator.3.2,
 fail.3, test.0.3, tock, coordinator.1.0,
 tock, tock, tock, tock, leader.2.3>
```

This can be interpreted as follows:

1. Nodes 2 and 3 fail.
2. Node 1 tests node 3; when it gets no response an election is started.
3. Both 2 and 3 revive shortly after they miss (through being failed) the election messages sent from 1.
4. 3 manages to tell 2 it is the coordinator before failing again.
5. 0 begins a test of 3.
6. By this time, 1 decides that, since it has not had any response to the election it started, it is the rightful coordinator. It therefore tells 0, which preempts the test it is doing.
7. The specification then gives the system time to settle (the `tock` events towards the end of the trace), though nothing of significance actually happens in this time.
8. We are left in a state where 0 and 1 believe 1 is the coordinator, and 2 believes that 3 is (though 3 has failed). Actually, of course, 2 ought to be agreed as coordinator since it is the highest-indexed live process.

The problem with the above is that all the individual node behaviors seem completely reasonable within the English description of the bully algorithm given earlier. The eventual conclusion of this study is that the algorithm, as described, is not

tolerant of a node failing in the middle of sending a sequence of `coordinator` messages to the nodes below it.

A solution is proposed in [75] which involves any node which is informed of the identity of the coordinator always testing that coordinator soon after. This works for all sizes of system that FDR has verified (up to seven nodes). It is by no means obvious, unfortunately, how one would create a proof by model checking that this corrected algorithm works for all N. (In more symmetric and simpler cases it is sometimes possible to justify arbitrary-sized networks, for example by combining induction and data independence as in [24–26].)

We might therefore draw the following lessons from this case study:

- Distributed algorithms frequently need to be made much more concrete before they can be model checked.
- Iterating a process algebra description can be an effective way of refining such an algorithm into a program.
- "Handshaking" process algebras can simulate networks where no output action is ever refused.
- Model checkers are exceptionally powerful tools for finding problems in distributed systems, but not so good at proving general results about arbitrary-sized systems.
- Sometimes it takes a larger system than you would think to find a problem with an algorithm: a number of people, having discovered the problem in the bully algorithm outlined above, have found "corrections" where four nodes were correct, but not five!

32.5.2 Active-Structure Control System (CWB)

Active structures include an embedded system that acts to limit structural vibration due to external excitations such as earthquakes or high winds. Typically, such structures include length-adjustable members, or *actuators*, that may be expanded or contracted to counteract the external forces applied to the structure. A process controller monitors sensors that measure the state of the structure and sends commands to the actuators when sensor readings indicate an undesirable state.

A major application of active-structure control systems involves earthquake-resistant buildings [82]. Earthquake damage often results less from the violence of movement than from the vibrations they induce in buildings. In particular, if an earthquake causes a structure to vibrate at its resonant frequency, then the structure becomes unstable and may collapse. Thus, to minimize vibration-induced earthquake damage, the natural frequencies of a structure should be located outside the frequency band of the seismic excitations produced by earthquakes.

An active-structural control system attempts to do this by sensing seismic excitations with a high sampling rate and changing the natural frequencies of the structure using the active members, with the particular method for changing these frequencies depending on the control algorithm used. Such control systems must satisfy

certain timing constraints on the activity of the actuators, since if the needed length-adjustments of the active members are not applied within the time bounds required by the control algorithm, then the structure may become unstable [2].

Several different control techniques have been proposed for active structures in the literature [1, 86]. In *pulse-control* approaches, the aim is to limit the vibratory displacement of structures near resonance by applying an opposing pulse at a higher frequency to "break up" the resonant forces. The major design variables are the time between pulse initiations, Δt_p, and the pulse duration, Δt. These values are determined by the natural frequencies of the system, the expected forcing functions, and the desired level of displacement control.

In [29], a case study involving the modeling and verification of a pulse-control system is described. The system uses a fault-detection and -recovery mechanism to cope with sensor and actuator failures, and was modeled first using the Modechart graphical notation and then translated into a real-time dialect of CCS. The resulting system contained over 10^{19} states and 13 parallel components; the specification was a simple timed CCS expression expressing timing constraints on inter-pulse durations. Using a compositional minimization strategy, the system was eventually proved equivalent to its specification using the Concurrency Workbench.

32.5.3 GNU i-Protocol (XSB)

The i-protocol, is part of the GNU uucp package available from the Free Software Foundation, and is used for file transfers over serial lines. The i-protocol sits on the uucp protocol stack; its purpose is to ensure ordered reliable duplex communication between two sites. At its lower interface, the i-protocol assumes unreliable (lossy) packet-based FIFO connectivity. At its upper interface, it provides *reliable* packet-based FIFO service. A distinguishing feature of the i-protocol is the sophisticated manner in which it attempts to minimize control-message and retransmission overhead. The GNU uucp package also contains the g- and j-protocols, which are variants of the i-protocol.

A problem with the i-protocol, GNU uucp version 1.04, was first noticed by Gene Stark[14] while trying to transfer large files from a remote computer to his home PC over a modem line. In particular, it appeared that, under certain message-loss conditions, the protocol would enter a "confused" state and eventually drop the connection. In order to diagnose this problem, we extracted an abstract version of the i-protocol from its source code, consisting of approximately 1500 lines of C code. We formalized this abstraction of the protocol in VPL (Value Passing Language), the input language of the Concurrency Factory specification and verification toolset [18].

The VPL source of the i-protocol was then subjected to a series of model-checking experiments using the Concurrency Factory's local model checker for the

[14]Eugene Stark is a Professor of Computer Science at Stony Brook University.

modal mu-calculus [68]. This led us to the root of the problem: a livelock that occurs when a particular series of message losses drives the protocol into a state where the communicating parties enter into a cycle of fruitless message exchanges without any packets being delivered to the upper-layer entities. Seeing no progress, the two sides close the connection, which must then be re-established. If the communication line is sufficiently noisy, or if one of the sides is slow in emptying communication buffers, say due to disk waits leading to buffer overflows, the chances of this scenario recurring are high, and can result in extremely poor performance.

Using the Concurrency Factory's diagnostic facility, we were able to pinpoint and subsequently "patch" the bug in the VPL code. The fix to the protocol consists of a simple change in the way negative acknowledgements are handled. The livelock error was fixed independently by Ian Taylor, the i-protocol's systematic case study conducted on the i-protocol original developer, in GNU uucp version 1.05.

Since the Concurrency Factory was applied to the problem, there has been a systematic case study conducted on the i-protocol using the Cospan, Murφ, Spin, and XMC verification tools. The i-protocol makes for a particularly interesting case study in protocol verification for several reasons. First, the version originally model checked there has a bug, i.e. the livelock error, and hence the protocol can be used to gauge a tool's ability to uncover errors of this nature. In this case, we are more interested in debugging or refutation than in verification.

Secondly, the size of the i-protocol's state space grows exponentially in the window size, and the entirety of this state space must be considered to verify that the protocol, with the livelock error eliminated, is deadlock- and livelock-free. Also, the i-protocol is an asynchronous, low-level software system equipped with a number of optimizations aimed at minimizing control-message and retransmission overhead. These optimizations further add to the protocol's complexity.

Thirdly, because of the i-protocol's inherent complexity, a novice tool user would immediately encounter difficulties in trying to analyze the protocol, due to the fact that the size of a system's state space is in general exponential in the size of the system's specification. This phenomenon is referred to as *state-space explosion*.

It is therefore imperative that certain modeling guidelines be followed when specifying the i-protocol in the input language of a model checker, to limit the effects of state-space explosion. Such guidelines are usually tool-specific and require a detailed knowledge of the tool's modeling language if they are to be deployed effectively. An informed choice of tool run-time options is also essential. Similarly, the results of our case study show that state-space explosion can be further curtailed by applying certain general-purpose abstraction techniques, several of which are identified in [43].

32.6 Conclusions

In this chapter, we have seen that the process algebras CSP, CCS, and ACP provide notations that are particularly well adapted to describing systems that interact by

(usually synchronous) message passing. We have both given examples of this type of system and referenced many practical applications of these ideas. The descriptions of such systems in process algebras are frequently efficient in terms of state spaces, and the model-checking technology developed for these theories can exploit the strong compositional properties they have.

The three process algebras we discussed have both been extended in various ways and encompass far more theory than we have had space to mention here. All three have real-time extensions [5, 70, 88], all of which were alluded to in this chapter. All three have been combined with probability (sometimes jointly with time), for example in [6, 36, 53].

The programming language occam [51], introduced to support the inmos Transputer in the 1980s, can be regarded as an imperative version of CSP, on which it was based.

The seminal calculus of mobility, the π-calculus [60, 79], is an extension of CCS, and of course has itself been extended in various ways. There was a reluctance to repeat, for mobile calculi, the multiplication that had earlier occurred in process algebras, but there has recently been interest in looking at mobility through the eyes of occam [85] and CSP [75].

Any serious development of a process algebra will inevitably have led to someone trying to develop verification technology for it, or at the very least investigating how to do this. Such technology is usually a form of model checking and its extensions described in other chapters of the present volume, including timed, symbolic, stochastic, and software model checking.

References

1. Abdel-Rohman, M., Leipholz, H.H.E.: Structural control by pole assignment method. Eng. Mech. **104**(5), 1157–1175 (1978)
2. Agrawal, A.K., Fujino, Y., Bhartia, B.K.: Instability due to time delay and its compensation in active control of structures. Earthq. Eng. Struct. Dyn. **22**(3), 211–224 (1993)
3. Armstrong, P., Goldsmith, M.H., Lowe, G., Ouaknine, J., Palikareva, H., Roscoe, A.W., Worrell, J.: Recent developments in FDR. In: Madhusudan, P., Seshia, S. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 7358, pp. 699–704. Springer, Heidelberg (2012)
4. Armstrong, P., Lowe, G., Ouaknine, J., Roscoe, A.W.: Model checking timed CSP. In: Voronkov, A., Korovina, M. (eds.) HOWARD-60. A Festschrift on the Occasion of Howard Barringer's 60th Birthday, pp. 13–33. EasyChair, Manchester (2014)
5. Baeten, J.C.M., Bergstra, J.A.: Real time process algebra. Form. Asp. Comput. **3**(2), 142–188 (1991)
6. Baeten, J.C.M., Bergstra, J.A., Smolka, S.A.: Axiomatizing probabilistic processes: ACP with generative probabilities. Inf. Comput. **121**(2), 234–255 (1995)
7. Baeten, J.C.M., Weijland, W.P.: Process Algebra. Cambridge Tracts in Computer Science, vol. 18. Cambridge University Press, Cambridge (1990)
8. Basu, S., Smolka, S.A.: Model checking the Java metalocking algorithm. ACM Trans. Softw. Eng. Methodol. **16**(3) (2007)
9. Bergstra, J.A., Klop, J.W.: Algebra of communicating processes with abstraction. Theor. Comput. Sci. **37**, 77–121 (1985)

10. Bergstra, J.A., Ponse, A., Smolka, S.A. (eds.): Handbook of Process Algebra. Elsevier, Amsterdam (2001)
11. Blom, S., van de Pol, J., Weber, M.: LTSmin: distributed and symbolic reachability. In: Touili, T., Cook, B., Jackson, P. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 6174, pp. 354–359. Springer, Heidelberg (2010)
12. Brookes, S.D., Hoare, C.A.R., Roscoe, A.W.: A theory of communicating sequential processes. J. ACM **31**(3), 560–599 (1984)
13. Brookes, S.D., Roscoe, A.W.: An improved failures model for communicating processes. In: Brookes, S.D., Roscoe, A.W., Winskel, G. (eds.) Proceedings of the Pittsburgh Seminar on Concurrency. LNCS, vol. 197, pp. 281–305. Springer, Heidelberg (1985)
14. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching-time temporal logic. In: Kozen, D. (ed.) Proceedings of the Workshop on Logic of Programs, Yorktown Heights. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1981)
15. Cleaveland, R.: On automatically explaining bisimulation inequivalence. In: Clarke, E.M., Kurshan, R.P. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 531, pp. 364–372. Springer, Heidelberg (1991)
16. Cleaveland, R., Hennessy, M.C.B.: Testing equivalence as a bisimulation equivalence. In: Sifakis, J. (ed.) Automatic Verification Methods for Finite State Systems. LNCS, vol. 407, pp. 11–23. Springer, Heidelberg (1989)
17. Cleaveland, R., Hennessy, M.C.B.: Testing equivalence as a bisimulation equivalence. Form. Asp. Comput. **5**(1), 1–20 (1993)
18. Cleaveland, R., Lewis, P.M., Smolka, S.A., Sokolsky, O.: The Concurrency Factory: a development environment for concurrent systems. In: Alur, R., Henzinger, T.A. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 1102, pp. 398–401. Springer, Heidelberg (1996)
19. Cleaveland, R., Parrow, J., Steffen, B.U.: The Concurrency Workbench: a semantics-based tool for the verification of finite-state systems. Tech. Rep. ECS-LFCS-89-83, Department of Computer Science, University of Edinburgh (1989)
20. Cleaveland, R., Parrow, J., Steffen, B.U.: A semantics-based tool for the verification of finite-state systems. In: Proceedings of the Ninth IFIP Symposium on Protocol Specification, Testing and Verification. North-Holland, Amsterdam (1989)
21. Cleaveland, R., Parrow, J., Steffen, B.U.: The Concurrency Workbench: a semantics-based tool for the verification of concurrent systems. ACM Trans. Program. Lang. Syst. **15**(1), 36–72 (1993)
22. Cleaveland, R., Sims, S.: Generic tools for verifying concurrent systems. Sci. Comput. Program. **42**(1), 39–47 (2002)
23. Colouris, G., Dollimore, J., Kindberg, T.: Distributed Systems, Concepts and Design. Addison-Wesley, Reading (1994)
24. Creese, S.J., Roscoe, A.W.: TTP: a case study in combining induction and data independence. Tech. Rep. PRG-TR-1-99, Oxford University Computing Laboratory (1999)
25. Creese, S.J., Roscoe, A.W.: Verifying an infinite family of inductions simultaneously using data independence and FDR. In: Wu, J., Chanson, S.T., Gao, Q. (eds.) Proceedings of FORTE/PSTV'99, pp. 437–452. Springer, Heidelberg (1999)
26. Creese, S.J., Roscoe, A.W.: Data independent induction over stuctured networks. In: Arabnia, H.R. (ed.) Proceedings of PDPTA 2000. CSREA (2000)
27. Dams, D., Grumberg, O.: Abstraction and abstraction refinement. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
28. Du, X., McDonnell, K.T., Nanos, E., Ramakrishna, Y.S., Smolka, S.A.: Software design, specification, and verification: lessons learned from the Rether Case Study. In: Johnson, M. (ed.) Proceedings of the Sixth Internationaal Conference on Algebraic Methodology and Software Technology (AMAST). LNCS, vol. 1349, pp. 185–198. Springer, Heidelberg (1997)
29. Elseaidy, W.M., Cleaveland, R., Baugh, J.W. Jr.: Modeling and verifying active structural control systems. Sci. Comput. Program. **29**(1–2), 99–122 (1997)

30. Emerson, E.A., Lei, C.L.: Efficient model checking in fragments of the propositional mu-calculus. In: Symp. on Logic in Computer Science (LICS), pp. 267–278. IEEE, Piscataway (1986)
31. Garcia-Molina, H.: Elections in a distributed computing system. IEEE Trans. Comput. **31**(1), 48–59 (1982)
32. Gibson-Robinson, T., Armstrong, P., Boulgakov, A., Roscoe, A.W.: FDR3: a modern refinement checker for CSP. In: Abraham, E., Mavelund, K. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 8413, pp. 187–201 (2014)
33. Gibson-Robinson, T., Roscoe, A.W.: FDR into the cloud. In: Welch, P.H., et al. (eds.) Proceedings of Communicating Process Architectures (CPA). Open Channel Publishing (2014)
34. Glabbeek, R.J.V., Weijland, W.P.: Branching time and abstraction in bisimulation semantics. J. ACM **43**(3), 555–600 (1996)
35. Groote, J.F., Mousavi, M.R.: Modeling and Analysis of Communicating Systems. MIT Press, Cambridge (2014)
36. Hansson, H., Jonsson, B.: A calculus for communicating systems with time and probabilities. In: Proceedings 11th Real-Time Systems Symposium (RTSS), pp. 278–287. IEEE, Piscataway (1990)
37. Hennessy, M.C.B., Milner, R.: Algebraic laws for nondeterminism and concurrency. J. ACM **32**(1), 137–161 (1985)
38. Henzinger, M.R., Henzinger, T.A., Kopke, P.W.: Computing simulations on finite and infinite graphs. In: 36th Annual Symposium on Foundations of Computer Science, pp. 453–462. IEEE, Piscataway (1995)
39. Henzinger, T.A., Manna, Z., Pnueli, A.: What good are digital clocks? In: International Colloquium on Automata, Languages and Programming (ICALP). LNCS, vol. 623, pp. 545–558. Springer, Heidelberg (1992)
40. Hoare, C.A.R.: Communicating sequential processes. Commun. ACM **21**(8), 666–677 (1978)
41. Hoare, C.A.R.: A model for communicating sequential processes, tech. monograph PRG-22, Programming Research Group, Oxford University Computing Laboratory (1981)
42. Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall, New York (1985)
43. Holzmann, G.J.: Designing executable abstractions. In: Proceedings of the Second Workshop on Formal Methods in Software Practice (FMSP), pp. 103–108. ACM, New York (1998)
44. Hopcroft, P.J., Broadfoot, G.: Combining the box structure development method and CSP for software development. Electron. Notes Theor. Comput. Sci. **128**(6), 127–144 (2005)
45. Hunt, H.B., Rosenkrantz, D.J., Szymanski, T.G.: On the equivalence, containment, and covering problems for the regular and context-free languages. J. Comput. Syst. Sci. **12**(2), 222–268 (1976)
46. Hwong, Y.L., Keiren, J.J.A., Kusters, V.J.J., Leemans, S., Willemse, T.A.C.: Formalising and analysing the control software of the Compact Muon Solenoid experiment at the Large Hadron Collider. Sci. Comput. Program. **78**(12), 2435–2452 (2013)
47. Kanellakis, P.C., Smolka, S.A.: CCS expressions, finite state processes, and three problems of equivalence. Inf. Comput. **86**(1), 43–68 (1990)
48. Kozen, D.: Results on the propositional μ-calculus. Theor. Comput. Sci. **27**(3), 333–354 (1983)
49. Lawrence, J.: Practical application of CSP and FDR in software design. In: Abdallah, A.E., Jones, C.B., Sanders, J.W. (eds.) Communicating Sequential Processes. The First 25 Years. LNCS, vol. 3525, pp. 151–175. Springer, Heidelberg (2005)
50. Lazic, R.S.: A semantic study of data independence with applications to model checking. Ph.D. thesis, University of Oxford (1999)
51. Limited, I.: Occam Programming Manual. Prentice Hall, New York (1984)
52. Liu, Y., Sun, J., Dong, J.S.: PAT 3: an extensible architecture for building multi-domain model checkers. In: Proceedings of the 22nd International IEEE Symposium on Software Reliability Engineering (ISSRE), pp. 190–199. IEEE, Piscataway (2011)

53. Lowe, G.: Probabilistic and prioritized models of timed CSP. Theor. Comput. Sci. **138**(2), 315–352 (1995)
54. Lowe, G.: Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In: Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 1055, pp. 147–166. Springer, Heidelberg (1996)
55. Lowe, G.: Casper: a compiler for the analysis of security protocols. In: Proceedings of the 10th Computer Security Foundations Workshop (CSFW), pp. 18–30. IEEE, Piscataway (1997)
56. Lu, S., Smolka, S.: Model checking the secure electronic transaction (SET) protocol. In: Proceedings of the 7th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), pp. 358–364. IEEE, Piscataway (1999)
57. Luttik, B.: What is algebraic in process theory? In: Proceedings of the Workshop "Essays on Algebraic Process Calculi" (APC 25). Electronic Notes in Theoretical Computer Science, vol. 162, pp. 227–231 (2006)
58. Milner, R.: A Calculus of Communicating Systems. LNCS, vol. 92. Springer, Heidelberg (1980)
59. Milner, R.: Communication and Concurrency. International Series in Computer Science. Prentice Hall, New York (1989)
60. Milner, R., Parrow, J., Walker, D.J.: A calculus of mobile processes. Inf. Comput. **100**(1), 1–40 (1992)
61. Ouaknine, J.: Discrete analysis of continuous behaviour in real-time concurrent systems. Ph.D. thesis, University of Oxford (2000)
62. Ouaknine, J., Palikareva, H., Roscoe, A.W., Worrell, J.: Static livelock analysis in CSP. In: Katoen, J.P., König, B. (eds.) Intl. Conf. on Concurrency Theory (CONCUR). LNCS, vol. 6901, pp. 389–403. Springer, Heidelberg (2011)
63. Paige, R., Tarjan, R.E.: Three partition refinement algorithms. SIAM J. Comput. **16**(6), 973–989 (1987)
64. Palikareva, H., Ouaknine, J., Roscoe, A.W.: SAT-solving in CSP trace refinement. Sci. Comput. Program. **77**(10), 1178–1197 (2012)
65. Peleska, J.: Applied formal methods—from CSP to executable hybrid specifications. In: Abdallah, A.E., Jones, C.B., Sanders, J.W. (eds.) Communicating Sequential Processes. The First 25 Years. LNCS, vol. 3525, pp. 293–320. Springer, Heidelberg (2005)
66. Ploeger, B.: Analysis of ACS using mCRL2. Tech. Rep. CS–09–11, Technische Universiteit Eindhoven (2009)
67. Plotkin, G.: A structural approach to operational semantics. Tech. Rep. DAIMI FN-19, Computer Science Department, Aarhus University (1981)
68. Ramakrishna, Y.S., Smolka, S.A.: Partial-order reduction in the weak modal mu-calculus. In: Mazurkiewicz, A., Winkowski, J. (eds.) Intl. Conf. on Concurrency Theory (CONCUR). LNCS, vol. 1243, pp. 5–24. Springer, Heidelberg (1997)
69. Ramakrishnan, C.R., Ramakrishnan, I.V., Smolka, S.A., et al.: XMC: a logic-programming-based verification toolset. In: Emerson, E.A., Sistla, A.P. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 1855, pp. 576–580. Springer, Heidelberg (2000)
70. Reed, G.M., Roscoe, A.W.: A timed model for communicating sequential processes. Theor. Comput. Sci. **58**(1–3), 249–261 (1988)
71. Roscoe, A.W.: Unbounded non-determinism in CSP. J. Log. Comput. **3**(2), 131–172 (1993)
72. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice Hall, New York (1997)
73. Roscoe, A.W.: Seeing beyond divergence. In: Abdallah, A.E., Jones, C.B., Sanders, J.W. (eds.) Communicating Sequential Processes. The First 25 Years. LNCS, vol. 3525, pp. 15–35. Springer, Heidelberg (2005)
74. Roscoe, A.W.: CSP is expressive enough for π. In: Jones, C.B., Roscoe, A.W., Wood, K.R. (eds.) Reflections on the Work of C.A.R. Hoare, pp. 371–404. Springer, Heidelberg (2010)
75. Roscoe, A.W.: Understanding Concurrent Systems. Springer, Heidelberg (2010)
76. Roscoe, A.W., Hopcroft, P.J.: Slow abstraction through priority. In: Liu, Z., Woodcock, J., Zhu, H. (eds.) Theories of Programming and Formal Methods. LNCS, vol. 8051, pp. 326–345. Springer, Heidelberg (2013)

77. Roscoe, A.W., Huang, J.: Checking noninterference in timed CSP. Form. Asp. Comput. **25**(1), 3–35 (2013)
78. Roscoe, A.W., Wu, Z.: Verifying statemate statecharts using CSP and FDR. In: Liu, Z., He, J. (eds.) Proceedings of Formal Methods and Software Engineering (FMSE). LNCS, vol. 4260, pp. 324–341. Springer, Heidelberg (2006)
79. Sangiorgi, D., Walker, D.J.: The pi-Calculus: A Theory of Mobile Processes. Cambridge University Press, Cambridge (2003)
80. Scattergood, J.: The semantics and implementation of machine-readable CSP. Ph.D. thesis, University of Oxford (1998)
81. Schneider, S.A.: Concurrent and Real-Time Systems. Wiley, New York (2000)
82. Soong, T.T.: Active Structural Control. Longman, New York (1990)
83. Steffen, B.: Characteristic formulae. In: Ausiello, G., Dezani-Ciancaglini, M., Rocca, S.R.D. (eds.) Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP). LNCS, vol. 372, pp. 723–732. Springer, Heidelberg (1989)
84. Walker, D.J.: Bisimulation and divergence in CCS. In: Symp. on Logic in Computer Science (LICS), pp. 186–192. IEEE, Piscataway (1988)
85. Welch, P.H., Barnes, F.R.M.: Communicating mobile processes. In: Abdallah, A.E., Jones, C.B., Sanders, J.W. (eds.) Communicating Sequential Processes. The First 25 Years. LNCS, vol. 3525, pp. 175–210. Springer, Heidelberg (2005)
86. Yang, J.N., Akbarpour, A., Ghaemmaghami, P.: New control algorithms for structural control. Eng. Mech. **113**(9), 1369–1386 (1987)
87. Yantchev, J.T.: ARC—a tool for efficient refinement and equivalence checking for CSP. In: IEEE Second Int. Conf. on Algorithms and Architectures for Parallel Processing (ICAPP), pp. 68–75. IEEE, Piscataway (1996)
88. Yi, W.: CCS + time = an interleaving model for real-time systems. In: Albert, J.L., Monien, B., Rodríguez-Artalejo, M. (eds.) International Colloquium on Automata, Languages and Programming (ICALP). LNCS, vol. 510, pp. 217–228. Springer, Heidelberg (1991)
89. Zakiuddin, M.I., Moffat, N., O'Halloran, C.M., Ryan, P.Y.A.: Chasing events to certify a critical system. Tech. rep., UK Defence Evaluation and Research Agency (1998)

Index

Symbols
2cl solver, 248
3-valued semantics, 19, 410, 831
ε (silent transition), 1012
μ-calculus, 11, 79, 229, 390, 568, 661, 766, 872, 927, 1113, 1184
  model checking, 887
  proof system, 901
  quantitative, 913, 1140
  relation to MSOL, 904
  satisfiability, 896
π-calculus, 1167
τ (internal action), 182, 362, 1152
ω-automaton, *see* Automaton
ω-regular language/property, 109, 154, 291, 559, 798, 864, 922, 970, 1013, 1133
✓ (successful termination event), 1167

A
ABC tool, 866
Abstract domain, 292, 406, 504, 566, 593, 671, 1142
Abstract interpretation, 5, 295, 399, 465, 494, 576, 632, 670, 785, 1087, 1142
Abstract state, 165, 293, 399, 442, 465, 494, 674, 721, 831, 1076
Abstract state machine, 75
Abstraction, 2, 17, 80, 154, 162, 265, 292, 310, 347, 385, 424, 447, 494, 574, 614, 655, 687, 752, 776, 809, 831, 1075
  acceleration, 698, 1142
  Cartesian, 229, 293, 428, 465, 515, 835
  convex hull, 1026
  corner-point, 1030
  counter, 720
  counterexample guided abstraction refinement (CEGAR), 17, 375, 402, 476, 526, 593, 866, 1075, 1082, 1181
  data, 575, 674, 831
  data-type, 93
  discrete, 20, 1076
  existential, 293, 399
  finite-chain, 594
  logical, 1086
  monotonic, 703
  of hybrid automaton, 1075
  predicate, 18, 292, 305, 385, 427, 447, 460, 466, 472, 478, 481, 515, 673, 712
  proof-based, 422
  refinement, 17, 277, 375, 402, 411, 424, 471, 526, 593, 637, 677, 787, 819, 991, 1082
  region, 1002
  soundness, 162, 388
  transition predicate, 466, 478, 952
Accellera standard, 781
Accepting condition, 108, 122, 124, 940, 1013
Accepting end component, 28, 989
Ackermann reduction, 331
ACL2 prover, 15, 487, 667, 866
ACP (Algebra of Communicating Processes), 1149, 1172
Active testing, 622
Active-structure control system, 1188
Adversary knowledge, 737
Alignment operator, 809
Alloy verification language, 84
Alt-Ergo solver, 306
Alternating
  automaton, *see* Automaton, alternating
  depth, 139, 878
  hierarchy, 899, 1115
  removal, 139
Ample set, 174
Antecedent failure, 8, 841

E.M. Clarke et al. (eds.), *Handbook of Model Checking*,
DOI 10.1007/978-3-319-10575-8

Antichain, 947, 1121
Apollo verification tool, 632
AppVerifier tool, 627
AProVe prover, 487, 533
Architecture description language, 82
Ariane 5 bug, 780
ARMC model checker, 486
ART verification tool, 1027
Assume-guarantee reasoning/rule, 15, 62, 349, 362, 375, 603, 716, 842, 953, 1100
  circular, 372
Assumption, *see* Assume-guarantee reasoning
Assumption generation, 367, 375
Astree verification tool, 785
Asynchronous circuit, 154, 173
Asynchronous composition/product, 88, 227, 350
Athena model checker, 751
ATL (Alternating-time Temporal Logic), 907, 935
AT&T network collapse, 780
Authentication, 740
Automaton, 107
  alternating, 136, 895
  Büchi, 13, 48, 108, 142, 159, 235, 280, 356, 487, 556, 694, 816, 907, 936, 943, 1119
  co-Büchi, 122
  complement, 112, 364, 1014
  containment, 132, 1002, 1015, 1064
  control-flow, 426, 501
  emptiness, *see* Language, emptiness
  intersection, *see* Language, intersection
  generalized Büchi, 122, 565
  modal, 892, 896, 903
  Muller, 123
  parity, 123, 898, 921, 1133
  Rabin, 122, 986
  Streett, 123
  typeness, 124
  union, 110
  universality, 132
  weak alternating, 138
Auxiliary variable, 359
AVISPA model checker, 749

B

BACH tool, 1064
Backtracking, 180, 200, 248, 252, 254, 314, 619
Backward analysis, 499
Bakery mutual-exclusion protocol, 673, 710
Bandera verification tool, 637
Barrier certificate, 1087
BDD, *see* Binary decision diagram
Beaver solver, 306
Bebop model checker, 243, 542
Behavioral congruence, 1161
Behavioral equivalence, 568, 1010, 1142, 1150
Behavioral pre-congruence, 1155, 1161
Bekic principle, 878
BerkMin solver, 248
BFS, *see* Breadth-first search
Binary decision diagram, 16, 37, 81, 179, 191, 219, 222, 248, 267, 277, 360, 405, 517, 542, 656, 721, 766, 848, 928
  apply algorithm, 197
  complement edge, 198
  for non-Boolean function, 206
  image computation, 224
  multi-terminal, 207, 966
  ordered, 195, 197, 1112, 2010
  partitioned, 205, 226
  variable ordering, 202
  zero-suppressed, 204
Biological application, 21, 77, 686, 964, 991
Bisimilarity, 162, 385, 394, 396, 891, 991, 1010, 1119
  approximate, 1083
  axiomatization, 1161
  branching, 182, 1185
  checking, 1011, 1176
  invariance, 891, 904
  logical characterization, 1157
  time-abstracted, 1007
  timed, 1010
  weak, 182, 1159
Bit-blasting, 81, 287, 323
BitBlaze testing tool, 632
Bit-state hashing, 84, 168, 621
Bit vector, 287, 322, 1181
BLAST model checker, 406, 431, 486, 500, 569
Bloom filter, 168
BMC, *see* Bounded model checking
Bohne verifier, 486
Boogie verification language, 84
Boolean formula, 16, 194, 222, 249, 279, 350, 466, 577, 679, 779, 797, 833, 888, 1112
Boolean program, 239, 293, 541, 637
Boolean satisfiability, 16, 81, 209, 247, 659, 1183
  eager encoding, 330
  model enumeration, 267
Boolector solver, 306
Bounded model checking, 16, 93, 278, 283, 286, 311, 406, 421, 501, 533, 576, 625, 675, 744, 766, 810, 1027, 1058, 1182

Bounded prover, 438
Bounded-session model checking, 732
Bounded synthesis, 946
Boyer–Moore theorem prover, 764
BPA (Basic Process Algebra), 1172
Branching time, 12, 52, 385, 500, 568, 580, 661, 771, 797, 965, 1021, 1022, 1116
BRB BDD package, 200
Breadth-first search, 155, 179, 210, 229, 464, 509, 1024, 1181
Broadcast, 578
Brutus model checker, 732
Büchi
  automaton, *see* Automaton, Büchi
  game, 928
Bully algorithm, 1185

C
Cache coherence protocol, 80, 685, 784
CAD, *see* Computer-aided design
Calfuzzer tool, 623
CaRet (Temporal Logic of Calls and Returns), 562
CatchConv testing tool, 632
CATG testing tool, 632
CAV, *see* Computer-aided verification
CAV Award, 766
Cayley trick, 1076
CBMC model checker, 285, 533
CCS (Calculus of Communicating Systems), 78, 574, 1151
  finite, 1161
  regular, 1163
CDCL, *see* Conflict-driven clause learning
CEGAR, *see* Abstraction, counterexample-guided abstraction refinement
Centaur technology, 866
Certificate, 4, 332, 525
CFA, *see* Automaton, control-flow
Chaff solver, 248, 766
Channel property, 755
Chase operator, 1181
Chess verification tool, 623, 1028
Choice operator, 369, 1152, 1165
CL-Atse protocol analyzer, 732
Clausal validity problem, 663
Clause learning, 254
Clock, 19, 81, 208, 286, 805, 935
Clock constraint, 19, 1003, 1013, 1124
Clock operator, 805
Clock region, *see* Region
Clock variable, 208, 1002, 1049, 1124
Closed system, 88
Clustering, 229, 1074
CMC model checker, 621
CMP (Chip Multi-Processor) router, 90
CMP method for parameterized verification, 727
CNF, *see* Conjunctive normal form
Co-Büchi automaton, *see* Automaton, co-Büchi
Co-Büchi condition/objective, 122, 925, 1134
Co-Büchi game, 928
CodeSonar, 785
Column transducer, 699
Communication channel, 601, 755, 964
Communication merge, 1174
Communication operation, 575
Communication pattern, 595, 637
Compactness, 658
Compassion, *see* Fairness
Compatibility checking, 922
Completeness, 354, 1153
  for abstraction framework, 412
  for assume-guarantee rule, 365
  for bisimulation equivalence, 1162
  for bounded programs, 285
  for consequence finding, 439
  for Hoare logic, 668
  for propositional logic, 658
  for propositional μ-calculus, 901
  for theory solver, 327, 330
  refutational, 328
  threshold, 289
Complete partial order, 1169
Composition of Boolean functions, 193
Composition of program analyses, 510
Composition operator, 668, 1151
  alternative, 1172
  parallel, 363, 369, 1005, 1152, 1165
  relational, 457
  sequential, 1166, 1172
Compositional modeling, 78
Compositional reasoning/verification, 345, 605, 709, 718, 857, 1099, 1180
Compositionality, 754
Compression, 1180
  lossless, 168
Compromised agent, 737
Computational soundness, 753
Computer-aided design, 769
Computer-aided verification, 1, 764
Concolic testing, 627
Concrete state, 166, 399, 502, 673
Concurrency Factory, 1189
Concurrency Workbench, 1183
Concurrent data structure, 589, 607, 953

Concurrent game, 934, 1139
Concurrent software/program, 16, 98, 286, 345, 573, 615, 633
Configurable program analysis, 493, 501, 511
Conflict analysis, 254, 315
Conflict clause, 316
Conflict-directed reachability, 678
Conflict-driven clause learning, 252, 313
Conflict set, 314
Conformal equivalence checker, 781
Congruence closure, 317
Conjunctive normal form, 249, 266, 280, 287, 315
Constant propagation, 498, 512, 520
Constraint satisfiability, 310, 317
Constraint solving, 422, 527, 625, 639, 744
Containment test, *see* Automaton, containment
Context-bounded model checking, 170
Contextual locking, 592
Continuous-set approximation, 1066
Continuous-set representation, 1070
Continuous successor, 1059, 1061, 1065, 1081
Continuous time, 19, 21, 78, 97, 992, 1084
Control-flow automaton, *see* Automaton, control-flow
Controller synthesis, 923, 951, 993, 1002, 1027, 1056, 1084, 1137
Coq theorem prover, 667
Correctness witness, *see* Certificate
Corruption model, 754
COSPAN model checker, 108, 1190
Cost-bounded operator, 976, 983, 984, 989
Cost function, 267, 967, 975
Cost-optimal schedule, 1029
Counterexample, 3, 8, 12, 100, 173, 180, 279, 289, 360, 368, 472, 525, 593, 652, 678, 1099
  feasible, 17, 294, 375, 401, 403, 411, 480, 595, 638, 675, 715, 1082
  lasso, 16, 404, 478, 676
  spurious, *see* feasible
Counting operator, 800
Counting property, 799, 1035
Coverability, 691, 1129
Coverity verification tool, 785
CPA, *see* Configurable program analysis
CPAchecker, 486, 501, 518, 533
CPAlien verification tool, 501
CPAtiger, 501
Craig interpolation, *see* Interpolation
CREST testing tool, 632
Crowds communication system, 757
Cryptographic complexity, 732
Cryptographic equational reasoning, 746
Cryptographic protocol, 18, 727, 1183
CSAT solver, 248
CSeq verification tool, 533
CSP (Communicating Sequential Processes), 78, 1163
  theoretical, 1164
CTA tool, 486
CTL (Computation Tree Logic), 10, 53, 181, 220, 378, 391, 567, 574, 817, 832, 906, 964, 1002
  fair, 220, 233
  first-order, 55
  model checking, 54, 231, 233
  Probabilistic, *see* PCTL
  satisfiability, 57
  translation to μ-calculus, 907
  weighted, 1031
CTL*, 7, 28, 63, 67, 181, 391, 567, 796, 902
  model checking, 69, 567
  satisfiability, 70
  translation to μ-calculus, 907
CuDD BDD package, 243, 928
Cumulated cost, 968, 1033
CUTE (Concolic Unit Testing Engine) tool, 632
CVC Theorem Prover, 306, 632
CWB (concurrency workbench), 1188
Cycle detection, 160, 176, 322, 557

D

DART (Directed Automated Random Testing), 627
Dash verification tool, 638
Data-flow analysis, 5, 493, 496, 504, 542, 605
Data-flow model, 78
Data independence, 155, 1150, 1188
Data-type reduction, 714
DBM, *see* Difference bound matrix
DDD, *see* Difference decision diagram
Deadlock, 63, 77, 153, 174, 356, 573, 615, 618, 695, 936, 1157
Deductive generalization, 421
Deductive verification, 45, 177, 653
Delayed theory combination, 327
Depth-first search, 155, 175, 464, 509, 634, 677, 752, 1181
  double, 180
  nested, 160
Derivation, 255, 423
Deterministic automaton, 109, 938, 1133
Determinization, 70, 120, 130, 377, 561, 802, 913, 935, 943, 1122, 1177
DFS, *see* Depth-first search
Diagnostics, 2, 789, 1178

Diamond compression, 1181
Difference bound matrix, 1024
Difference decision diagram, 208, 1026
Difference logic, 321, 1027
Differentiable, 1054
Dirac distribution, 967
Discounted game, 890, 933
Discounted objective, 926, 993
Discrete successor, 1063, 1070, 1081
Discrete time, 18, 78, 757, 805, 974, 1003, 1056, 1182
Disjunctive normal form, 249, 312, 895
Disjunctive well-foundedness, 466
Disolver solver, 632
Distinguishing formula, 1178
Distributed leadership-election protocol, 1185
Divergence, 754, 1014, 1167
  closed, 1170
  pre-order, 1184
  respecting, 1170
  strict, 1168
Division property of the μ-calculus, 902
DNF, *see* Disjunctive normal form
Dolev–Yao model, 731
DPLL (Davis–Putnam–Logemann–Loveland) algorithm, 212, 248, 295, 316, 787
DPLL(T), 269, 289, 316
Dynamic logic, 6617989061099
Dynamic partial-order reduction, *see* Partial-order
Dynamic precision adjustment, 521
Dynamic symbolic execution, 627
Dynamic test generation, 626, 627
Dynamic variable reordering, 202

E

EDA, *see* Electronic design automation
Edge-lean algorithm, 187
Edge-triggered design, 806
EGT testing tool, 632
Electronic design automation, 765
Elimination order, 435
Ellipsoid, 1070
Else verification tool, 1027
Embedded C-code, 165
Emptiness, *see* Language, emptiness
Encapsulation, *see* Hiding
Ended operator, 802
Energy constraint, 1032
Energy game, 933
Energy objective, 926
Environment model, 85, 772
Equality with uninterpreted functions, 317, 515
Equational axiom, 1151
Equational certificate, 1089
Equational theory, 746, 1172
Equivalence checking, 19, 286, 306, 568, 780, 1177
Equivalence query, 367
Equivalence relation, 165, 185, 290, 317, 394, 699, 1010, 1155
Error witness, *see* Counterexample
ESBMC verification tool, 533
EUF, *see* Equality with uninterpreted functions
Event-clock automaton, 1015
Event of cryptographic protocol, 735
Event structure, 188
Evolution domain, 1052, 1069, 1079
Exclusive or, 212, 746
EXE testing tool, 432
Existentially quantified Horn solver, 487
Existentially quantified transition, 704
Expected cost, 965
Explanation generation, 315
Explicit-heap analysis, 516, 523
Explicit state, 15, 37, 153, 175, 219, 360, 441, 513, 604, 656, 743, 785, 1049, 1181
Expressiveness, 5, 27, 78, 145, 305, 414, 562, 580, 660, 765, 817, 871, 946, 1001

F

Failure (process algebra), 1168
Failures-divergences model, 1167
Failures refinement, 1169
Fair computation problem, 37, 102, 233, 356, 547
  generalized, 555
Fair discrete system, 30, 390
Fairness, 12, 31, 89, 234, 390, 655, 694, 936, 1022
Falsification, 2, 411, 750, 777
Farkas' lemma, 440, 527
FAST verification tool, 756
FDR (Failures-Divergences Refinement) model checker, 731, 1181
Feaver verification tool, 637
Finite-model property, 663, 901
Finite-variant property, 747
Finitely nondeterministic CPS, 1170
First-match property, 800
First-order logic, 16, 31, 46, 175, 284, 307, 423, 517, 565, 628, 661, 710, 764, 798, 904, 1120
Fixed point, 11, 38, 213, 289, 349, 391, 441, 465, 496, 542, 587, 661, 742, 865, 878, 1022, 1098, 1113, 1170
  greatest, 873
  least, 353, 873
  operator, 873

Fixpoint-Analysis Machine, 568
FormalCheck model checker, 773
Formality equivalence checker, 781
FormalPro equivalence checker, 781
ForSpec verification tool, 108
FORTE verification system, 831
Forward analysis, 525
Forward flow, 433
Forward search, 742
Fourier–Motzkin elimination, 318, 1051
FrankenBit verification tool, 533
FSM, *see* State machine, finite
F-Soft model checker, 284, 486
FuncTion verification tool, 533
Fusion property, 801

G
Galois connection, 671, 832
Game semantics, 880
Generalization, 155
Generalized trajectory logic, 865
German's protocol, 713
Global condition, 686
GNU i-protocol, 1189
Goto operator, 800
GR(1) Generalized Reactivity-1, 951, 1134
Graph game, 18, 882, 924, 1131
  deterministic, 1133
  non-zero-sum, 953
  objective, 925
  play, 924
  probabilistic, 1137
GRASP solver, 248, 766
Ground term/formula, 328
Guarantee language/property, 45, 346, 349, 362, 364, 372
Guarded fixed-point logic, 912

H
Handshaking, 1164
Hardware description language, 82, 281, 806
Hardware/software co-verification, 277, 286
Hash collision, 169
Hash-compact, 169
Hausdorff distance, 1067
Heap analysis, 516
Helicopter model, 97
Hennessy–Milner Logic (HML), 661, 1157
Hiding, 80, 167, 1165
Higher-order logic (HOL), 665, 860
Hoare logic, 425, 563, 667, 858, 912
HOL (Higher-Order Logic) prover, 857
Homomorphic encryption, 746
Honest thread, 739
Horn clause, 434, 752
Houdini annotation assistant, 486
HSF solver, 486
HSolver model checker, 1098
Hybrid automaton, 1028, 1052, 1061, 1065
  non-linear, 1072
  rectangular, 1055, 1126
Hybrid system, 19, 29, 78, 108, 655, 778, 1003, 1048
Hybrid verification, 778, 1099
Hybridization, 1075
HyTech model checker, 1027, 1063, 1127

I
IBMC, *see* Interpolant-based model checking
IC (Integrated Circuit) design, 769
IC3, *see* Iterative inductive strengthening
IEEE 754 standard, 851
IEV verification tool, 778
IKE (Internet Key Exchange) protocol, 732
Image computation, 224, 289, 405, 434, 454, 1051
Image-finite, 1158
Impact model checker, 486
Implication graph, 251
Independence relation, 175
Inductive invariant, 290, 349, 424, 457, 625
Inductive transition invariant, 459
Inference rule, 333, 496, 656, 746, 858, 1151
Infinite-state
  game, 935
  Markov chain, 971
  system, 31, 108, 310, 405, 673, 686, 733, 1002, 1115
  transducer, 699
Informative prefix, 815
Initial condition, 30, 350, 442, 530, 1052
Initial configuration, 134, 497, 546, 579, 690
Input-space decomposition, 851
Instantaneous cost, 974
Integer arithmetic, 319, 431, 1100
Integer difference logic, 322
Interface, 1, 80, 192, 316, 364, 606, 632, 767, 1102, 1165
  equality, 327
  generation, 377
  variable, 325
Interference, 348, 576, 636
Interleaving semantics, 88, 173, 286, 718
Interpolation, 220, 265, 291, 312, 421, 485, 527, 659, 766, 901
Interpretation
  abstract, *see* Abstract interpretation
  Sigma-interpretation, 308, 388, 656

Interrogator verification tool, 731
Intersection, *see* Language, intersection
Interval property checking, 832
Introduction order, 437
Intruder deduction, 744
Invariant, 44, 60, 231, 279, 311, 349, 426, 712, 906, 1003, 1052, 1125
  differential, 1085
  generation, 424, 525, 605
Isabelle theorem prover, 667
ISO/IEC 9798 standard, 733
Iterated structure, 908
Iteration algorithm, 504, 933
Iteration order, 509
Iterative inductive strengthening, 292

J
Jakstab verification tool, 501
Jalangi testing tool, 632
Java Pathfinder tool, 621
jCUTE testing tool, 633
Join, 496, 503, 525
Jump, 1052
Justice, *see* Fairness

K
Karp–Miller algorithm, 691
Kerberos protocol, 727
KeYmaera verification tool, 1099
k-induction, 277, 311, 675
KISS verification tool, 756
KLEE testing tool, 632
k-liveness, 281
Klocwork verification tool, 785
Kripke structure, 6, 30, 75, 102, 141, 221, 230, 278, 388, 500
Kronos model checker, 1027
Kudzu testing tool, 632

L
L* learning algorithm, 366
Labeling function, 6, 30, 102, 237, 388, 577, 938
LAMBDA theorem prover, 765
Language
  emptiness, 132, 180, 561, 580, 693, 816, 887, 1012, 1064, 1112
  intersection, 111, 159, 180, 556, 592, 707, 1014
Lasso, 12, 160, 404, 478, 676
Lattice, 503, 670, 833, 1113
Layered theory solver, 329
Lazy abstraction, 442
Lazy data type, 750
Learning, 22, 366, 529
Left-merge, 1173
Level of abstraction, 80, 406, 526, 788, 1167
Level-set method, 1101
Linear constraint, 1051
Linear integer arithmetic, 319
Linear programming, 982, 1064, 1140
Linear real arithmetic, 318
Linear term, 1051
Linear time, 63, 500, 661, 1135
Linearization, 173, 622, 1074
Liquid Types, 486, 667
LiQuor model checker, 990
Literal deduction, 315
Live-variable analysis, 499
Livelock, 1190
Liveness, 12, 43, 61, 88, 158, 280, 357, 578, 694
  component, 803
  verification, 158
LLBMC, 533
Local assertion, 351
Local proof, 357, 422, 431
Local symmetry, 361
Local variable, 710, 820
Localization, 401, 422, 736, 766, 787
Location, 1052
Location analysis, 511
Lock, 578, 634, 688
  acquisition history, 581
  causality graph, 586
  chain, 589
  nested, 581
  pattern, 591
Logic programming, 1184
Logical certificate, 1090
Logical characterization, 1150, 1157
LoopFrog verification tool, 294, 486
Lossy channel, 601, 992
lp_solve solver, 632
LTA tool, 486
LTL (Linear Temporal Logic), 12, 42, 158, 279, 556, 797, 907, 942, 985
  extensions, 46, 796
  first-order, 46
  indexed, 579
  model checking, 15, 50, 146, 235, 987
  satisfiability, 50, 146
  synthesis, 942
  translation to automaton, 13, 142
  translation to μ-calculus, 907
LTSA verification tool, 369
LTSmin, 1185

M
MaceMC model checker, 623
Magellan verification tool, 778
Magic model checker, 486
Markov chain, 12, 207, 757, 970
  continuous-time, 990
Markov decision process, 18, 935, 964, 967, 971, 980, 987
  continuous-time, 992
  product, 988
Mars Orbiter bug, 780
Master method, 530
MathSAT solver, 306
Maude-NPA protocol analyzer, 748
Mazurkiewicz trace, 603
mCRL2 toolset, 1184
MDP, *see* Markov decision process
Mealy machine, 89, 936
Mean-payoff game, 933, 934
Mean-payoff objective, 926
Meet over all paths, 498
Membership query, 367
Memory consistency, 604
Memory model, 604
Merge operator, 16, 508, 514, 1139, 1173
Message pattern, 738
Message sequence chart, 78, 728
Minimization, 121, 229, 259, 280, 378, 398, 1150, 1185
MiniSAT solver, 248
Minkowski sum, 1062
Mixed integer and real arithmetic, 321
Moby model checker, 1027
Mocha model checker, 378
Modal automaton, *see* Automaton, modal
Modal logic, 7, 28, 660, 874, 891, 1112, 1151
Modal simulation, 406
Modal transition system, 407
Modality, 393, 660, 798, 876, 964, 978, 1018
Model-based quantifier instantiation, 329
Model-based testing, 4, 639
Model-driven verification, 167
Model measuring, 21
Modeling, 18, 29, 75, 77, 79, 82, 86, 88, 94, 173, 247, 322, 544, 666, 1001
  challenge, 3, 18, 83, 776, 1048
  formalism/language, 75, 76, 78, 82, 282, 614, 1101, 1149
MoDist verification tool, 623
Modular reasoning, 377
Modularity, 80, 429
Monadic second-order logic (MSOL), 139, 720, 764, 798, 904, 1017
  over nested words, 562
Monitor automaton, 519
Monotonic fixed-point approach, 525
Monotonic transition relation, 706
Moore machine, 89, 936
MOPED model checker, 542
MSOL, *see* Monadic second-order logic
MTBDD, *see* Binary decision diagram, multi-terminal
MTL (Metric Temporal Logic), 1018
Muller
  automaton, *see* Automaton, Muller
  game, 932
Multi-core algorithm, 170
Multi-index temporal-logic formula, 578
Multi-phase acyclic pushdown network, 597
Murφ model checker, 84, 486, 1190
Murphi verification system, 84, 486, 713, 732, 784, 1190
Mutual exclusion, 34, 388, 574, 654, 704, 1021, 1130
  protocol, 101, 354, 369, 687

N
Near-neighbor communication, 695
Needham–Schroeder public-key protocol, 731
Negation normal form, 1114
Negative acknowledgement, 1190
Nelson–Oppen method, 327, 664
Nested word, 557
  automaton, 558
  temporal logic, 562
Net-list, 282
Network, 78, 90, 154, 195, 347, 591, 596–598, 686, 728, 964, 1181
Next operator, 42, 392, 563, 796, 808, 980
Next-state function, 837
NIL process, 1152
Non-determinism, 63, 83, 94, 108, 113, 136, 167, 182, 325, 519, 561, 605, 615, 714, 747, 816, 887, 923, 964, 1055, 1121, 1156
Non-interference, 348, 709, 756
Non-linear dynamics, 1074
Non-monotonic approach, 528
Non-trace property, 755
Normalization, 1024, 1181
NPA (NRL Protocol Analyzer), 748
NPATRL logic, 748
NTAB solver, 248
Numerical simulation, 22, 1058
Nuprl theorem prover, 667
NuSMV model checker, 108, 243

O
OBDD, *see* Binary decision diagram, ordered
Obligation language/property, 45
Observational congruence, 1161
Observational equivalence, 756, 1159
Observer automaton, 518
Occam (programming language), 1193
Occam's razor, 422
Off-line guessing, 756
OFMC verification tool, 750
Omega test, 320
On-the-fly, 15, 133, 159, 180
One-to-one correspondence, 823
Open system, 69, 88, 108, 378
openSMT solver, 306
Orion verification tool, 518
Owicki–Gries method, 348

P
Pairwise reachability, 587
Pan model checker, 154
Parameterized systems, 77, 655, 685, 953
Parameterized verification, 686, 709
Parametric representation, 851
Parasoft verification tool, 785
Parity automaton, *see* Automaton, parity
Parity game/tree automaton, 130, 568, 767, 872, 930, 934, 1134
Parse tree, 495, 993
Partial correctness, 563, 652, 841
Partial-information game, 935
Partial order, 173, 503
Partial-order reduction, 15, 84, 153, 173, 174, 603, 621, 745, 991, 1150
  for CTL, 181
  for LTL, 176
  for process algebra, 182
Partial-order semantics, 174
Partition, 1175, 1177
PASS model checker, 991
Past operator, 54, 802
Path, 6, 31, 102, 110, 162, 195, 236, 279, 284, 390, 474, 477
  constraint, 277, 624, 1064
  encoding, 284
  event, 976
  reductiveness, 426
  slicing, 294
PathCrawler testing tool, 637
PCTL (Probabilistic CTL), 757, 964
  model checking, 980
PCTL*, 993
PDL (Propositional Dynamic Logic), 906
Pentium FDIV bug, 780
Permissive, 364, 376
Persistence language/property, 45
PET verification tool, 636
Peterson's mutual-exclusion protocol, 34, 62, 654
Petri net, 78, 687, 1129
PEX testing tool, 632
Phase-portrait approximation, 1078
PHAVER verifier, 1100
PicoSAT solver, 248
Piecewise affine dynamics, 1064
Piecewise constant dynamics, 1060
Piterman's construction, 130, 943
Policy, *see* Strategy
Policy iteration, *see* Strategy improvement
Polyhedron, 1051
Polynomial approximation, 1075
Polynomial constraint, 1050
Polynomial term, 1050
Polyspace verification tool, 785
Polytope, 1051
POSIT solver, 248
Positional strategy, *see* Strategy
Post operator, 37
Post-closure, 454
Post-condition, 454, 652
Pre operator, 37, 586
Pre-condition, 652
Pre-congruence, 1155
Precision, 507
Predator verification tool, 533
Predicate, 307, 461, 1050
Predicate analysis, 515
Predicated lattice, 520
Predictive analysis, 592
Predictive model, 606
Pre-emptive context bounding, 606
Prefixing, 1152
Prenex normal form, 662
Pre-order, 1151
Presburger arithmetic, 320, 663
PRISM model checker, 757, 990
Probabilistic inference, 528
Probabilistic model checking, 757, 963
  approximate, 991
Probabilistic system, 18, 79, 756, 967, 1137
  metric, 1142
Probability distribution, 967
Probability measure, 970
Probability space, 970
ProbDiVinE model checker, 990
Process algebra (PA), 19, 78, 182, 1151
Process transformation, 1177
Program, 284, 425, 448

Program (*cont.*)
asynchronous, 153
interrupt-driven, 100
message-passing, *see* with rendezvous
multi-threaded, 33, 348, 573, 621
procedural, 541
shared-variables, *see* multi-threaded
with rendezvous, 592
Program analysis, 504
Program counter, 34, 284, 448, 502
Program repair, 953, 1137
Program representation, 500
Program sketching, 953, 1137
Promela language, 107, 637, 991
Proof generalization, 441
Proof-rule-based approach, 528
Proof system, 333, 422, 658, 901
Propositional logic, 247, 280, 334, 656, 833, 980, 1181
interpretation, 656
proof system, 657
Protocol role, 734
ProVerif protocol analyzer, 732, 752
PSL (Property Specification Language), 283, 795
Pulse control, 1189
Pure term/formula, 327
Purify tool, 627
Push-down automaton/system, 78, 238, 545, 575, 785, 910, 935, 992
communicating, 592
interacting, 579
Push-down model checking, 238
Push-down network, 592
multi-phase acyclic, 597
lossy-channel, 601
PVS (Prototype Verification System), 666

Q

QBF (Quantified Boolean formula), 268
Quantified variable, 818
Quantifier elimination, 278, 291, 328, 436, 663, 1063, 1127
Quantifier-free interpolation property, 334
Quantitative abstraction refinement, 21, 991
Quantitative objective, 926
Quantitative verification, 21, 1142
QuartzFormal equivalence checker, 781
Quotienting, 21, 700

R

Rabbit model checker, 1027
Rabin
automaton, *see* Automaton, Rabin
game, 925
Randomized strategy, *see* Strategy
Ranking function, 358, 480, 653
RAPTURE model checker, 991
RCP (Relational Coarsest Partitioning), 1175
Reachability, 32
Reaching definition, 513
Reactive module, 78
Reactive programming, 83
Reactive synthesis, 921
Reactivity language/property, 45
Ready set, 1169
Real arithmetic, 318
Realizability, 924
Real-time logic, 1018
Real-time system, 18, 79, 1003
Recurrence language/property, 45
Recurrence solving, 529
RED verification tool, 1027
Refinement
abstraction, *see* Abstraction-refinement
checking, 1177
ordering, 1169
Refiner, 424
Refusal, 1169
Refutation, 277, 430
Region
algebra, 1115
automaton, 1007
equivalence, 1007
Regular expression, 696, 797
weak/strong, 802
Regular model checking, 695, 1142
Relabeling, *see* Renaming
Relational composition/product, 193, 226, 457, 764
Relsat solver, 248
Rely-guarantee reasoning, 354
Renaming, 221, 1152
Rendezvous, 578, 686
Repetition operator, 799
Reset, hardware, 810
Residual, 846
Resolution, 193, 251, 291, 334, 432
Restriction, 192, 1152
Reverse flow, 433
Robustness, 1037, 1083
Romeo, 1027
RTL (Register Transfer Level) language, 287, 768
RuleBase model checker, 773
Run-time scheduler, 619
Run-time verification, 638

S
Safe, 364, 452
Safety, 12, 43, 84, 88, 155, 231, 279, 447, 451, 454, 460, 690, 698, 711, 1116
  accidental, 816
  automaton, 936
  component, 804
  game, 928, 1034
  of hybrid automaton, 1054
  intentional, 816
  invariant, 12, 43, 60, 350, 426
  objective, 925
  operator, 391
  pathological, 817
  verification, 1054, 1097
Safraless synthesis, 946
Safra's construction, 114, 130, 943
SAGE testing tool, 633
SAL (Symbolic Analysis Laboratory) language, 83
SAL model checker, 84, 676
SART verification tool, 1027
SAT, *see* Boolean satisfiability
SAT-based model checking, 212, 277, 431, 786
  completeness, 289
  image computation, 289
SAT engine/solver, 247, 252, 253, 264
  branching heuristics, 263
  clause minimization, 259
  incremental, 265
  lazy data structure, 261
  non-chronological backtracking, 254
  search restart, 263
SATABS model checker, 294, 406, 486
Satisfiability, 50, 56, 69, 146, 657, 896
Satisfiability modulo theories, 305
SAT-MC, 732, 744, 750
SATO solver, 248
Saturation, 552, 567
  differential, 1094
Scenario, 78
Scheduler, *see* Strategy
Scyther protocol analyzer, 751
Second-order logic, 46, 562, 906
Secrecy, 739
Secrecy pattern, 751
Semantics, 46, 102, 308, 562, 615, 656, 738, 821, 872, 976, 986, 1054, 1113
  3-valued, *see* 3-valued semantics
  axiomatic, 1161
  denotational, 496, 1163
  operational, 689, 1004, 1151
Semi-group property, 1066
Semi-lattice, 503
Separation logic, 607
Separation of concerns, 824
Sequence, 834, 836
Sequence interpolant, 431
Sequential circuit, 834
Sequential consistency, 604
Sequential equivalence checking, 286, 781
Sequentialization, 605
Session of cryptographic protocol, 748
Shannon, Claude, 766
Shannon expansion, 192
Shape analysis, 532
Signature, 307, 861
Similarity, 529, 891, 1011, 1119
  checking, 1177
Simplex method, 319
Simplify theorem prover, 293
Simulation, 83, 294, 831, 1058, 1175
Simulation relation, 394
  time-abstracted, 1007
  timed, 1010
SixthSense model checker, 773
SKIP process, 1165
SLAB model checker, 486
SLAM model checker, 292, 312, 406, 434, 486, 541, 638, 785
Sleep set, 183
SLEC equivalence checker, 781
Small-model property, 899
Small-steps method, 779
Smallest sufficient model, 155
SMART model checker, 202
Smash model checker, 638
SML (simple modeling language), 85
SMT (Satisfiability Modulo Theories) solver, 247, 288, 305
  incremental, 314
  lazy, 267, 313
SMT-COMP, 306
SMT-EXEC, 306
SMTInterpol solver, 306
SMT-LIB, 307
SMV model checker, 84, 637, 718, 776
Software model checking, 17, 79, 283, 312, 434, 506, 604, 637
Software verification, 5, 100, 153, 188, 283, 401, 447, 493, 541, 866
SONOLAR solver, 306
Sort symbols, 307
SOS, *see* Structural operational semantics
Soundness, 162, 388, 524, 753
  for assume-guarantee rule, 347, 365
  for Hoare logic, 668
  for propositional logic, 658

refutational, 332
SpaceEx verification tool platform, 84, 1101
Spec# specification language, 84
SpecC verification tool, 294
Specialization, 155
Specification, 2, 27, 79, 107, 173, 219, 283, 378, 518, 556, 638, 795, 925, 1169
SPIN model checker, 84, 108, 159, 743
Splat testing tool, 632
Split invariant, 351
Split prover, 439
Splitting on demand, 330
SSA, *see* Static single assignment
SSL (Secure Sockets Layer) protocol, 727
Stability, 1084, 1168
Stable-failures model, 1169
Stably infinite, 326, 664
Star-free ω-regular language/property, 795
State compression, 168, 1180
State descriptor, 153
State explosion, *see* State-space explosion
State formula/predicate, 7, 31, 158, 232, 390, 975, 1021
State machine, 3, 75, 203, 229, 362, 542, 615, 947, 1154
  finite, 3, 78, 229, 362, 615, 948, 1155
  recursive, 542
State space
  explosion, 3, 80, 173, 345, 385, 621, 743, 966, 1025, 1190
  minimization, 1150
  reduction, 749, 1181
  search, 160, 620, 741
State vector, 165
Statecharts, 78, 786, 1183
Stateless search, 621
Statemate tool, 1183
Static equivalence, 756
Static single assignment form, 284, 430
Static test generation, 625
Statistical model checking, 21, 991, 1037
STE, *see* Symbolic trajectory evaluation
STE deductive system, 858
STE model checking, 844
Stochastic game, 19, 890, 934, 992
STOP process, 1165
STP solver, 306, 632
Strategy, 882, 924, 1034
  deterministic, 1132
  finite-memory, 924
  improvement, 931
  memoryless, *see* Positional
  optimal, 890, 927, 981, 1002, 1140
  positional, 882, 924
  randomized, 1138
  simple, 971
  winning, 882, 927
Streett automaton, *see* Automaton, Streett
Streett condition/objective, 124, 952
Streett game, 931
Strengthening, 277, 510, 654, 712, 804, 858
Strong component, 804
Strong equivalence, 1156
Strong fairness, 30, 102
Strong regular expression, 802
Strongest post-condition, 350, 434, 516, 672
Strongest split invariant, 352
Structural contradiction, 804
Structural method, 15
Structural operational semantics, 1151, 1167
Stuttering, 39, 88, 159, 175
  bisimulation, 182
  equivalence, 175
Subset construction, 118, 377, 935, 1015, 1121
Substitution method, 529
Subterm convergence, 747
Subtyping, 496, 667
Suffix implication, 798
Summarization, 541, 604
Supertrace hashing, 169
Support function, 1072
Support of a distribution, 967
SVA (System Verilog Assertions), 781, 795, 1183
Symbolic algorithm/method, 16, 37, 229, 398, 675, 845, 927, 1023, 1123
Symbolic execution, 284, 377, 435, 527, 623, 676, 862
Symbolic indexing, 832
Symbolic model checking, 16, 153, 191, 219, 312, 378, 441, 625, 672, 720, 766, 786, 796, 833, 1111
  BDD-based, 210, 219, 656, 1124
  of timed automaton, 1024
Symbolic session generation, 750
Symbolic state, 229, 750, 786, 1059
Symbolic trajectory evaluation, 19, 787, 831
  generalized, 864
  relational, 860
Symbolic transition system, 87, 672
Symbiotic verification tool, 533
Symmetric, 162, 373, 709, 1157
Symmetric rule, 373
Symmetry, 361
Symmetry reduction, 15, 176, 621, 721, 787, 991, 1027
Synchronization operation, 605
Synchronous circuit, 90, 153, 1002

Synchronous composition/product, 52, 81, 159, 227, 350, 926, 987
Synchronous control system, 96
Synchronous language, 83
Synergy algorithm, 442, 486, 638, 680, 782, 1150
Synopsys Corporation, 859
Syntax tree, 222, 494
Synthesis, 20, 45, 77, 130, 282, 378, 527, 607, 656, 767, 872, 921, 1002, 1056, 1111
  symbolic, 1131
SystemC language, 82, 286, 788

T
T2 verification tool, 533
Tableau, 13, 57, 237, 565
Tamarin prover, 744, 752
TAME, 1027
TAN verification tool, 533
TAPALL verification tool, 1027
Taylor model, 1075
TCTL (Timed CTL), 1021
Technology transfer, 763
Template-based approach, 529
Template polyhedron, 1073
Temporal hierarchy, 43
Temporal logic, 6, 27, 795
  branching, 7, 181
  extended, 46
  linear, 42
  quantified, 818
  regular-expression-based, 797
  simple subset, 817
  weighted, 1031
Temporal tester, 50
Term of cryptographic protocol, 734
Termination, 356, 447, 547, 622, 1118
Test bench, 770
Test generation, 625, 676
Testing, 4, 613
Theorem proving, 5, 764
Theory, 305, 309, 663
  axiom, 309
  combination, 324, 664
  convex, 326
  interpretation, 309
  lemma, 333
  literal, 307
  matching, 328
  propagation, 315
  quantified, *see* Quantifier elimination
  satisfiability, 309
  theory, 307
  unsatisfiable core, 333
  validity, 309
Therac-25 bug, 780
Thread of cryptographic protocol, 738
Thread-modular verification, 607
Threader verification tool, 486, 533
Tic-tac-toe, 163
Time discretization, 1066, 1101
Time sampling, 809
Time successor of a polyhedron, 1061
Timed automaton, 208, 1003, 1124
  complement, 1014
  containment, *see* Automaton, containment
  emptiness, 1015
  model checking, 1019
  probabilistic, 992
  weighted, 1028
Timed CSP, 1172, 1182
Timed game, 935, 1034
Timed language, 1012
Timed word, 1003
Times, 1027
Token-passing protocol, 696
ToolboxLS (Level Set), 1101
Total cost, *see* Cumulated cost
Toyota braking problem, 782
TPTL (Timed Propositional Temporal Logic), 1018
Trace, 363, 737, 986
  equivalence, 185, 1167
  normal form, 184
  refinement, 1169
Traces model, 1169
Trajectory, 838
  assertion, 840
  evaluation, 787
  evaluation logic, 838
  formula, 838
Transducer, 697, 938
Transfer relation, 504
Transition
  invariant, 453
  predicate, 467
  probability function, 967
  relation, 6
  weak, 1159
Transition system, 2, 101, 310, 407, 1112
  fair, *see* Fair discrete system
  labeled, 6, 500
  rooted, 1156
  symbolic, 87
  well-quasi-ordered, 687
Tree automaton, 893, 939
Tree-model property, 891
Triggers operator, *see* Suffix implication

Truncated path, 809
Tseitin encoding, 287
TVLA tool, 486
Type checking, 494
Type logic, 495

U
UCLID verifier, 84, 331
UFO verification tool, 486, 533
Ultimate Automizer model checker, 486
Unfolding, 188, 431, 874, 879
Unified algorithm, 506
Uninterpreted function, 82, 309, 317, 431, 515, 627
Uninterpreted symbol, 309
Unique fixed-point induction, 1163
Unique implication point, 256
Unit clause rule, 250
Unit propagation, 250
Universal algebra, 1149
Universal tree automaton, 939
Universally quantified Horn solver, 486
Universally quantified transition, 704
Unsatisfiable core, 265, 333, 679
Until operator, 42, 796, 843, 906, 981, 986
UPC-Thrille tool, 623
Updated variable, 1052
UPPAAL model checker, 84, 1027, 1137
Upward closed set, 691, 1129
Urgent state, 1187
Utility of proof, 422

V
Vacuity, 12, 841
  hidden, 842
Valgrind tool, 627
Validity, 51, 309, 663, 907, 980
Value analysis, 513, 533
Value iteration, 933, 982
Variant function, 653
Variant narrowing, 747
Verics model checker, 1027
Verification condition, 624, 653
Verification game, 883
Verification language, 83
Verilog language, 82, 282, 769
VeriSoft tool, 620
veriT solver, 306
Verity equivalence checker, 781
VeSTA verification tool, 1027
VHDL language, 82, 282, 769
Viability, 38, 40
Visible content, 1159
Visibly pushdown automaton, 558
VOSS verification system, 858
VPL (Value Passing Language), 1189

W
WALi library, 542
Watched-literals data structure, 262
Weak aliveness, 740
Weak alternating automaton, *see* Automaton, weak alternating
Weak bisimulation, *see* Bisimilar, weak
Weak component, 804
Weak disagreement, 845
Weak fairness, 31, 89, 694
Weak MSOL, 905
Weak regular expression, 803
Weakening, 348, 436, 677, 852, 1128
Weakest assumption, 364
Weakest pre-condition, 294, 437, 483, 515, 669, 720
Well-founded relation, 453
Well-quasi-order, 687, 1016, 1128
Well-structured transition system, 1128
White-box fuzzing, 633
Widening, 507, 720, 1059
Wiggling, 845
Wolverine verification tool, 406, 486
Worst-case execution time, 21, 285
Wrapping effect, 1068

X
XMC model checker, 1184

Y
YAPA verification tool, 756
Yices solver, 306
Yogi verification tool, 441, 486, 638

Z
Z3 solver, 306, 632
Zapato theorem prover, 293
ZChaff solver, 279
ZDD, *see* Binary decision diagram, zero-suppressed
Zenoness, 1005
Zone, 1023, 1126
Zonotope, 1071
ZZ toolset, 866