

Isil Dillig
Serdar Tasiran (Eds.)

LNCS 11561

Computer Aided Verification

31st International Conference, CAV 2019
New York City, NY, USA, July 15–18, 2019
Proceedings, Part I

1
Part I



Springer Open

Commenced Publication in 1973

Founding and Former Series Editors:

Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board Members

David Hutchison

Lancaster University, Lancaster, UK

Takeo Kanade

Carnegie Mellon University, Pittsburgh, PA, USA

Josef Kittler

University of Surrey, Guildford, UK

Jon M. Kleinberg

Cornell University, Ithaca, NY, USA

Friedemann Mattern

ETH Zurich, Zurich, Switzerland

John C. Mitchell

Stanford University, Stanford, CA, USA

Moni Naor

Weizmann Institute of Science, Rehovot, Israel

C. Pandu Rangan

Indian Institute of Technology Madras, Chennai, India

Bernhard Steffen

TU Dortmund University, Dortmund, Germany

Demetri Terzopoulos

University of California, Los Angeles, CA, USA

Doug Tygar

University of California, Berkeley, CA, USA

More information about this series at <http://www.springer.com/series/7407>

Isil Dillig · Serdar Tasiran (Eds.)

Computer Aided Verification

31st International Conference, CAV 2019
New York City, NY, USA, July 15–18, 2019
Proceedings, Part I



Springer Open

Editors

Isil Dillig
University of Texas
Austin, TX, USA

Serdar Tasiran
Amazon Web Services
New York, NY, USA



ISSN 0302-9743

ISSN 1611-3349 (electronic)

Lecture Notes in Computer Science

ISBN 978-3-030-25539-8

ISBN 978-3-030-25540-4 (eBook)

<https://doi.org/10.1007/978-3-030-25540-4>

LNCS Sublibrary: SL1 – Theoretical Computer Science and General Issues

© The Editor(s) (if applicable) and The Author(s) 2019. This book is an open access publication.

Open Access This book is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this book are included in the book's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the book's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

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

Preface

It was our privilege to serve as the program chairs for CAV 2019, the 31st International Conference on Computer-Aided Verification. CAV 2019 was held in New York, USA, during July 15–18, 2019. The tutorial day was on July 14, 2019, and the pre-conference workshops were held during July 13–14, 2019. All events took place in The New School in New York City.

CAV is an annual conference dedicated to the advancement of the theory and practice of computer-aided formal analysis methods for hardware and software systems. The primary focus of CAV is to extend the frontiers of verification techniques by expanding to new domains such as security, quantum computing, and machine learning. This put CAV at the cutting edge of formal methods research, and this year’s program is a reflection of this commitment.

CAV 2019 received a very high number of submissions (258). We accepted 13 tool papers, two case studies, and 52 regular papers, which amounts to an acceptance rate of roughly 26%. The accepted papers cover a wide spectrum of topics, from theoretical results to applications of formal methods. These papers apply or extend formal methods to a wide range of domains such as concurrency, learning, and industrially deployed systems. The program featured invited talks by Dawn Song (UC Berkeley), Swarat Chaudhuri (Rice University), and Ken McMillan (Microsoft Research) as well as invited tutorials by Emina Torlak (University of Washington) and Ranjit Jhala (UC San Diego). Furthermore, we continued the tradition of Logic Lounge, a series of discussions on computer science topics targeting a general audience.

In addition to the main conference, CAV 2019 hosted the following workshops: The Best of Model Checking (BeMC) in honor of Orna Grumberg, Design and Analysis of Robust Systems (DARS), Verification Mentoring Workshop (VMW), Numerical Software Verification (NSV), Verified Software: Theories, Tools, and Experiments (VSTTE), Democratizing Software Verification, Formal Methods for ML-Enabled Autonomous Systems (FoMLAS), and Synthesis (SYNT).

Organizing a top conference like CAV requires a great deal of effort from the community. The Program Committee for CAV 2019 consisted of 79 members, a committee of this size ensures that each member has to review a reasonable number of papers in the allotted time. In all, the committee members wrote over 770 reviews while investing significant effort to maintain and ensure the high quality of the conference program. We are grateful to the CAV 2019 Program Committee for their outstanding efforts in evaluating the submissions and making sure that each paper got a fair chance.

Like last year’s CAV, we made artifact evaluation mandatory for tool submissions and optional but encouraged for the rest of the accepted papers. The Artifact Evaluation Committee consisted of 27 reviewers who put in significant effort to evaluate each artifact. The goal of this process was to provide constructive feedback to tool developers and help make the research published in CAV more reproducible. The Artifact Evaluation Committee was generally quite impressed by the quality of the artifacts,

and, in fact, all accepted tools passed the artifact evaluation. Among regular papers, 65% of the authors submitted an artifact, and 76% of these artifacts passed the evaluation. We are also very grateful to the Artifact Evaluation Committee for their hard work and dedication in evaluating the submitted artifacts.

CAV 2019 would not have been possible without the tremendous help we received from several individuals, and we would like to thank everyone who helped make CAV 2019 a success. First, we would like to thank Yu Feng and Ruben Martins for chairing the Artifact Evaluation Committee and Zvonimir Rakamaric for maintaining the CAV website and social media presence. We also thank Oksana Tkachuk for chairing the workshop organization process, Peter O’Hearn for managing sponsorship, and Thomas Wies for arranging student fellowships. We also thank Loris D’Antoni, Rayna Dimitrova, Cezara Dragoi, and Anthony W. Lin for organizing the Verification Mentoring Workshop and working closely with us. Last but not least, we would like to thank Kostas Ferles, Navid Yaghmazadeh, and members of the CAV Steering Committee (Ken McMillan, Aarti Gupta, Orna Grumberg, and Daniel Kroening) for helping us with several important aspects of organizing CAV 2019.

We hope that you will find the proceedings of CAV 2019 scientifically interesting and thought-provoking!

June 2019

Isil Dillig
Serdar Tasiran

Organization

Program Chairs

Isil Dillig The University of Texas at Austin, USA
Serdar Tasiran Amazon, USA

Workshop Chair

Oksana Tkachuk Amazon, USA

Publicity Chair

Zvonimir Rakamaric University of Utah, USA

Sponsorship Chair

Peter O'Hearn Facebook, USA

Fellowship Chair

Thomas Wies NYU, USA

CAV Award Committee

Natarajan Shankar
Pierre Wolper
Somesh Jha
Parosh Abdulla

SRI International, USA
Liege University, Belgium
University of Wisconsin, USA
Uppsala University, Sweden

Program Committee

Aws Albarghouthi	University of Wisconsin-Madison, USA
Jade Alglave	University College London, UK
Rajeev Alur	University of Pennsylvania, USA
Christel Baier	TU Dresden, Germany
Gilles Barthe	Max Planck Institute for Security and Privacy, Germany; IMDEA Software Institute, Spain
Osbert Bastani	University of Pennsylvania, USA
Josh Berdine	Facebook, USA
Per Bjesse	Synopsys Inc., USA
Nikolaj Bjorner	Microsoft, USA
Roderick Bloem	Graz University of Technology, Austria

Marc Brockschmidt	Microsoft, UK
Pavol Cerny	University of Colorado Boulder, USA
Swarat Chaudhuri	Rice University, USA
Wei-Ngan Chin	National University of Singapore
Adam Chlipala	Massachusetts Institute of Technology, USA
Hana Chockler	King's College London, UK
Eva Darulova	Max Planck Institute for Software Systems, Germany
Cristina David	University of Cambridge, UK
Dana Drachsler Cohen	ETH Zurich, Switzerland
Cezara Dragoi	Inria Paris, ENS, France
Constantin Enea	IRIF, University of Paris Diderot, France
Azadeh Farzan	University of Toronto, Canada
Grigory Fedyukovich	Princeton University, USA
Yu Feng	University of California, Santa Barbara, USA
Dana Fisman	Ben-Gurion University, Israel
Milos Gligoric	The University of Texas at Austin, USA
Patrice Godefroid	Microsoft, USA
Laure Gonnord	University of Lyon/Laboratoire d'Informatique du Parallélisme, France
Aarti Gupta	Princeton University, USA
Arie Gurfinkel	University of Waterloo, Canada
Klaus Havelund	Jet Propulsion Laboratory, USA
Chris Hawblitzel	Microsoft, USA
Alan J. Hu	The University of British Columbia, Canada
Shachar Itzhaky	Technion, Israel
Franjo Ivancic	Google, USA
Ranjit Jhala	University of California San Diego, USA
Rajeev Joshi	Automated Reasoning Group, Amazon Web Services, USA
Dejan Jovanović	SRI International, USA
Laura Kovacs	Vienna University of Technology, Austria
Burcu Kulahcioglu Ozkan	MPI-SWS, Germany
Marta Kwiatkowska	University of Oxford, UK
Shuvendu Lahiri	Microsoft, USA
Akash Lal	Microsoft, India
Stephen Magill	Galois, Inc., USA
Joao Marques-Silva	Universidade de Lisboa, Portugal
Ruben Martins	Carnegie Mellon University, USA
Ken McMillan	Microsoft, USA
Vijay Murali	Facebook, USA
Peter Müller	ETH Zurich, Switzerland
Mayur Naik	Intel, USA
Hakjoo Oh	Korea University, South Korea
Oded Padon	Stanford University, USA
Corina Pasareanu	CMU/NASA Ames Research Center, USA
Ruzica Piskac	Yale University, USA

Nir Piterman	University of Gothenburg, Sweden
Pavithra Prabhakar	Kansas State University, USA
Sylvie Putot	LIX, Ecole Polytechnique, France
Grigore Rosu	University of Illinois at Urbana-Champaign, USA
Dorsa Sadigh	Stanford University, USA
Roopsha Samanta	Purdue University, USA
Sriram Sankaranarayanan	University of Colorado, Boulder, USA
Koushik Sen	University of California, Berkeley, USA
Sanjit A. Seshia	University of California, Berkeley, USA
Natarajan Shankar	SRI International, USA
Rahul Sharma	Microsoft, USA
Natasha Sharygina	Università della Svizzera italiana (USI Lugano), Switzerland
Sharon Shoham	Tel Aviv University, Israel
Alexandra Silva	University College London, UK
Rishabh Singh	Google, USA
Anna Slobodova	Centaur Technology, USA
Marcelo Sousa	University of Oxford, UK
Cesare Tinelli	The University of Iowa, USA
Ufuk Topcu	University of Texas at Austin, USA
Caterina Urban	Inria, France
Margus Veanes	Microsoft, USA
Yakir Vizel	The Technion, Israel
Chao Wang	USC, USA
Georg Weissenbacher	Vienna University of Technology, Austria
Eran Yahav	Technion, Israel
Hongseok Yang	KAIST, South Korea

Artifact Evaluation Committee

Uri Alon	Technion, Israel
Yaniv David	Technion, Israel
Yufei Ding	University of California, Santa Barbara, USA
Yu Feng (Co-chair)	University of California, Santa Barbara, USA
Radu Grigore	University of Kent, UK
Saurabh Joshi	IIT Hyderabad, India
William Hallahan	Yale University, USA
Travis Hance	Carnegie Mellon University, USA
Marijn Heule	The University of Texas at Austin, USA
Antti Hyvärinen	University of Lugano, Switzerland
Alexey Ignatiev	Universidade de Lisboa, Portugal
Tianhan Lu	University of Colorado Boulder, USA
Ruben Martins (Co-chair)	Carnegie Mellon University, USA
Aina Niemetz	Stanford University, USA
Filip Nikšić	University of Pennsylvania, USA
Lauren Pick	Princeton University, USA

Sorawee Porncharoenwase	University of Washington, USA
Mathias Preiner	Stanford University, USA
Talia Ringer	University of Washington, USA
John Sarracino	University of California San Diego, USA
Xujie Si	University of Pennsylvania, USA
Calvin Smith	University of Wisconsin-Madison, USA
Caleb Stanford	University of Pennsylvania, USA
Miguel Terra-Neves	INESC-ID/IST, Universidade de Lisboa, Portugal
Jacob Van Geffen	University of Washington, USA
Xinyu Wang	The University of Texas at Austin, USA
Wei Yang	The University of Texas at Dallas, USA

Mentoring Workshop Organizing Committee

Loris D'Antoni (Chair)	University of Wisconsin, USA
Anthony Lin	Oxford University, UK
Cezara Dragoi	Inria, France
Rayna Dimitrova	University of Leicester, UK

Steering Committee

Ken McMillan	Microsoft, USA
Aarti Gupta	Princeton, USA
Orna Grunberg	Technion, Israel
Daniel Kroening	University of Oxford, UK

Additional Reviewers

Sepideh Asadi	Emanuele D'Osualdo
Lucas Asadi	Nicolas Dilley
Haniel Barbosa	Marko Dilley
Ezio Bartocci	Bruno Dutertre
Sam Bartocci	Marco Eilers
Suda Bharadwaj	Cindy Eilers
Erdem Biyik	Yotam Feldman
Martin Biyik	Jerome Feret
Timothy Bourke	Daniel Feret
Julien Braine	Mahsa Ghasemi
Steven Braine	Shromona Ghosh
Benjamin Caulfield	Anthony Ghosh
Eti Chaudhary	Bernhard Gleiss
Xiaohong Chaudhary	Shilpi Goel
Yinfang Chen	William Goel
Andreea Costea	Mirazul Haque
Murat Costea	Ludovic Henrio

Andreas Henrio	Rong Palmskog
Antti Hyvärinen	Daejun Park
Duligur Ibeling	Brandon Paulsen
Rinat Ibeling	Lucas Paulsen
Nouraldin Jaber	Adi Yoga Prabawa
Swen Jacobs	Dhananjay Raju
Maximilian Jacobs	Andrew Raju
Susmit Jha	Heinz Riener
Anja Karl	Sriram Sankaranarayanan
Jens Karl	Mark Sankaranarayanan
Sean Kauffman	Yagiz Savas
Ayrat Khalimov	Traian Florin Serbanuta
Bettina Khalimov	Fu Serbanuta
Hillel Kugler	Yahui Song
Daniel Larraz	Pramod Subramanyan
Christopher Larraz	Rob Subramanyan
Wonyeol Lee	Sol Swords
Matt Lewis	Martin Tappler
Wenchao Lewis	Ta Quang Tappler
Kaushik Mallik	Anthony Vandikas
Matteo Marescotti	Marcell Vazquez-Chanlatte
David Marescotti	Yuke Vazquez-Chanlatte
Dmitry Mordvinov	Min Wen
Matthieu Moy	Josef Widder
Thanh Toan Moy	Bo Widder
Victor Nicolet	Haoze Wu
Andres Noetzli	Zhe Xu
Abraham Noetzli	May Xu
Saswat Padhi	Yi Zhang
Karl Palmskog	Zhizhou Zhang

Contents – Part I

Automata and Timed Systems

Symbolic Register Automata	3
<i>Loris D'Antoni, Tiago Ferreira, Matteo Sammartino, and Alexandra Silva</i>	
Abstraction Refinement Algorithms for Timed Automata	22
<i>Victor Roussanaly, Ocan Sankur, and Nicolas Markey</i>	
Fast Algorithms for Handling Diagonal Constraints in Timed Automata.	41
<i>Paul Gastin, Sayan Mukherjee, and B. Srivathsan</i>	
Safety and Co-safety Comparator Automata for Discounted-Sum Inclusion.	60
<i>Suguman Bansal and Moshe Y. Vardi</i>	
Clock Bound Repair for Timed Systems	79
<i>Martin Kölbl, Stefan Leue, and Thomas Wies</i>	
Verifying Asynchronous Interactions via Communicating Session Automata	97
<i>Julien Lange and Nobuko Yoshida</i>	

Security and Hyperproperties

Verifying Hyperliveness.	121
<i>Norine Coenen, Bernd Finkbeiner, César Sánchez, and Leander Tentrup</i>	
Quantitative Mitigation of Timing Side Channels	140
<i>Saeid Tizpaz-Niari, Pavol Černý, and Ashutosh Trivedi</i>	
Property Directed Self Composition.	161
<i>Ron Shemer, Arie Gurfinkel, Sharon Shoham, and Yakir Vizel</i>	
Security-Aware Synthesis Using Delayed-Action Games	180
<i>Mahmoud Elfar, Yu Wang, and Miroslav Pajic</i>	
Automated Hypersafety Verification	200
<i>Azadeh Farzan and Anthony Vandikas</i>	
Automated Synthesis of Secure Platform Mappings.	219
<i>Eunsuk Kang, Stéphane Lafourcade, and Stavros Tripakis</i>	

Synthesis

Synthesizing Approximate Implementations for Unrealizable Specifications	241
<i>Rayna Dimitrova, Bernd Finkbeiner, and Hazem Torfah</i>	
Quantified Invariants via Syntax-Guided Synthesis	259
<i>Grigory Fedyukovich, Sumanth Prabhu, Kumar Madhukar, and Aarti Gupta</i>	
Efficient Synthesis with Probabilistic Constraints	278
<i>Samuel Drews, Aws Albarghouthi, and Loris D'Antoni</i>	
Membership-Based Synthesis of Linear Hybrid Automata	297
<i>Miriam García Soto, Thomas A. Henzinger, Christian Schilling, and Luka Zelezniak</i>	
Overfitting in Synthesis: Theory and Practice	315
<i>Saswat Padhi, Todd Millstein, Aditya Nori, and Rahul Sharma</i>	
Proving Unrealizability for Syntax-Guided Synthesis	335
<i>Qinheping Hu, Jason Breck, John Cyphert, Loris D'Antoni, and Thomas Reps</i>	

Model Checking

BMC for Weak Memory Models: Relation Analysis for Compact SMT Encodings	355
<i>Natalia Gavrilenko, Hernán Ponce-de-León, Florian Furbach, Keijo Heljanko, and Roland Meyer</i>	
When Human Intuition Fails: Using Formal Methods to Find an Error in the “Proof” of a Multi-agent Protocol	366
<i>Jennifer A. Davis, Laura R. Humphrey, and Derek B. Kingston</i>	
Extending nuxmv with Timed Transition Systems and Timed Temporal Properties	376
<i>Alessandro Cimatti, Alberto Griggio, Enrico Magnago, Marco Roveri, and Stefano Tonetta</i>	
Cerberus-BMC: A Principled Reference Semantics and Exploration Tool for Concurrent and Sequential C	387
<i>Stella Lau, Victor B. F. Gomes, Kayvan Memarian, Jean Pichon-Pharabod, and Peter Sewell</i>	

Cyber-Physical Systems and Machine Learning

Multi-armed Bandits for Boolean Connectives in Hybrid System Falsification	401
<i>Zhenya Zhang, Ichiro Hasuo, and Paolo Arcaini</i>	
StreamLAB: Stream-based Monitoring of Cyber-Physical Systems	421
<i>Peter Faymonville, Bernd Finkbeiner, Malte Schledjewski, Maximilian Schwenger, Marvin Stenger, Leander Tentrup, and Hazem Torfah</i>	
VERIFAI: A Toolkit for the Formal Design and Analysis of Artificial Intelligence-Based Systems	432
<i>Tommaso Dreossi, Daniel J. Fremont, Shromona Ghosh, Edward Kim, Hadi Ravanbakhsh, Marcell Vazquez-Chanlatte, and Sanjit A. Seshia</i>	

The Marabou Framework for Verification and Analysis of Deep Neural Networks	443
<i>Guy Katz, Derek A. Huang, Duligur Ibeling, Kyle Julian, Christopher Lazarus, Rachel Lim, Parth Shah, Shantanu Thakoor, Haoze Wu, Aleksandar Zeljić, David L. Dill, Mykel J. Kochenderfer, and Clark Barrett</i>	

Probabilistic Systems, Runtime Techniques

Probabilistic Bisimulation for Parameterized Systems (with Applications to Verifying Anonymous Protocols)	455
<i>Chih-Duo Hong, Anthony W. Lin, Rupak Majumdar, and Philipp Rümmer</i>	
Semi-quantitative Abstraction and Analysis of Chemical Reaction Networks	475
<i>Milan Češka and Jan Křetínský</i>	
PAC Statistical Model Checking for Markov Decision Processes and Stochastic Games	497
<i>Pranav Ashok, Jan Křetínský, and Maximilian Weininger</i>	
Symbolic Monitoring Against Specifications Parametric in Time and Data	520
<i>Masaki Waga, Étienne André, and Ichiro Hasuo</i>	
STAMINA: STochastic Approximate Model-Checker for INfinite-State Analysis	540
<i>Thakur Neupane, Chris J. Myers, Curtis Madsen, Hao Zheng, and Zhen Zhang</i>	

Dynamical, Hybrid, and Reactive Systems

Local and Compositional Reasoning for Optimized Reactive Systems	553
<i>Mitesh Jain and Panagiotis Manolios</i>	
Robust Controller Synthesis in Timed Büchi Automata: A Symbolic Approach	572
<i>Damien Busatto-Gaston, Benjamin Monmege, Pierre-Alain Reynier, and Ocan Sankur</i>	
Flexible Computational Pipelines for Robust Abstraction-Based Control Synthesis	591
<i>Eric S. Kim, Murat Arcak, and Sanjit A. Seshia</i>	
Temporal Stream Logic: Synthesis Beyond the Bools	609
<i>Bernd Finkbeiner, Felix Klein, Ruzica Piskac, and Mark Santolucito</i>	
Run-Time Optimization for Learned Controllers Through Quantitative Games	630
<i>Guy Avni, Roderick Bloem, Krishnendu Chatterjee, Thomas A. Henzinger, Bettina Könighofer, and Stefan Pranger</i>	
Taming Delays in Dynamical Systems: Unbounded Verification of Delay Differential Equations	650
<i>Shenghua Feng, Mingshuai Chen, Naijun Zhan, Martin Fränzle, and Bai Xue</i>	
Author Index	671

Contents – Part II

Logics, Decision Procedures, and Solvers

Satisfiability Checking for Mission-Time LTL	3
<i>Jianwen Li, Moshe Y. Vardi, and Kristin Y. Rozier</i>	
High-Level Abstractions for Simplifying Extended String Constraints in SMT	23
<i>Andrew Reynolds, Andres Nötzli, Clark Barrett, and Cesare Tinelli</i>	
Alternating Automata Modulo First Order Theories	43
<i>Radu Iosif and Xiao Xu</i>	
Q3B: An Efficient BDD-based SMT Solver for Quantified Bit-Vectors	64
<i>Martin Jonáš and Jan Strejček</i>	
cvc4sy: Smart and Fast Term Enumeration for Syntax-Guided Synthesis	74
<i>Andrew Reynolds, Haniel Barbosa, Andres Nötzli, Clark Barrett, and Cesare Tinelli</i>	
Incremental Determinization for Quantifier Elimination and Functional Synthesis	84
<i>Markus N. Rabe</i>	

Numerical Programs

Loop Summarization with Rational Vector Addition Systems	97
<i>Jake Silverman and Zachary Kincaid</i>	
Invertibility Conditions for Floating-Point Formulas	116
<i>Martin Brain, Aina Niemetz, Mathias Preiner, Andrew Reynolds, Clark Barrett, and Cesare Tinelli</i>	
Numerically-Robust Inductive Proof Rules for Continuous Dynamical Systems	137
<i>Sicun Gao, James Kapinski, Jyotirmoy Deshmukh, Nima Roohi, Armando Solar-Lezama, Nikos Arechiga, and Soonho Kong</i>	
Icing: Supporting Fast-Math Style Optimizations in a Verified Compiler	155
<i>Heiko Becker, Eva Darulova, Magnus O. Myreen, and Zachary Tatlock</i>	
Sound Approximation of Programs with Elementary Functions	174
<i>Eva Darulova and Anastasia Volkova</i>	

Verification

Formal Verification of Quantum Algorithms Using Quantum Hoare Logic	187
<i>Junyi Liu, Bohua Zhan, Shuling Wang, Shenggang Ying, Tao Liu, Yangjia Li, Mingsheng Ying, and Naijun Zhan</i>	

SEC CSL: Security Concurrent Separation Logic.	208
<i>Gidon Ernst and Toby Murray</i>	

Reachability Analysis for AWS-Based Networks.	231
<i>John Backes, Sam Bayless, Byron Cook, Catherine Dodge, Andrew Gacek, Alan J. Hu, Temesghen Kahsai, Bill Kocik, Evgenii Kotelnikov, Jure Kukovec, Sean McLaughlin, Jason Reed, Neha Rungta, John Sizemore, Mark Stalzer, Preethi Srinivasan, Pavle Subotić, Carsten Varming, and Blake Whaley</i>	

Distributed Systems and Networks

Verification of Threshold-Based Distributed Algorithms by Decomposition to Decidable Logics.	245
<i>Idan Berkovits, Marijana Lazić, Giuliano Losa, Oded Padon, and Sharon Shoham</i>	

Gradual Consistency Checking	267
<i>Rachid Zennou, Ahmed Bouajjani, Constantin Enea, and Mohammed Erradi</i>	

Checking Robustness Against Snapshot Isolation	286
<i>Sidi Mohamed Beillahi, Ahmed Bouajjani, and Constantin Enea</i>	

Efficient Verification of Network Fault Tolerance via Counterexample-Guided Refinement.	305
<i>Nick Giannarakis, Ryan Beckett, Ratul Mahajan, and David Walker</i>	

On the Complexity of Checking Consistency for Replicated Data Types	324
<i>Ranadeep Biswas, Michael Emmi, and Constantin Enea</i>	

Communication-Closed Asynchronous Protocols	344
<i>Andrei Damian, Cezara Drăgoi, Alexandru Militaru, and Josef Widder</i>	

Verification and Invariants

Interpolating Strong Induction.	367
<i>Hari Govind Vediramana Krishnan, Yakir Vizel, Vijay Ganesh, and Arie Gurfinkel</i>	

Verifying Asynchronous Event-Driven Programs Using Partial Abstract Transformers	386
<i>Peizun Liu, Thomas Wahl, and Akash Lal</i>	
Inferring Inductive Invariants from Phase Structures	405
<i>Yotam M. Y. Feldman, James R. Wilcox, Sharon Shoham, and Mooly Sagiv</i>	
Termination of Triangular Integer Loops is Decidable	426
<i>Florian Frohn and Jürgen Giesl</i>	
AliveInLean: A Verified LLVM Peephole Optimization Verifier	445
<i>Juneyoung Lee, Chung-Kil Hur, and Nuno P. Lopes</i>	
Concurrency	
Automated Parameterized Verification of CRDTs	459
<i>Kartik Nagar and Suresh Jagannathan</i>	
What’s Wrong with On-the-Fly Partial Order Reduction.	478
<i>Stephen F. Siegel</i>	
Integrating Formal Schedulability Analysis into a Verified OS Kernel	496
<i>Xiaojie Guo, Maxime Lesourd, Mengqi Liu, Lionel Rieg, and Zhong Shao</i>	
Rely-Guarantee Reasoning About Concurrent Memory Management in Zephyr RTOS	515
<i>Yongwang Zhao and David Sanán</i>	
Violat: Generating Tests of Observational Refinement for Concurrent Objects.	534
<i>Michael Emmi and Constantin Enea</i>	
Author Index	547

Automata and Timed Systems



Symbolic Register Automata

Loris D’Antoni¹, Tiago Ferreira², Matteo Sammartino²(✉),
and Alexandra Silva²

¹ University of Wisconsin–Madison, Madison, WI 53706-1685, USA
loris@cs.wisc.edu

² University College London, Gower Street, London WC1E 6BT, UK
me@tiferrei.com, {m.sammartino,a.silva}@ucl.ac.uk

Abstract. Symbolic Finite Automata and Register Automata are two orthogonal extensions of finite automata motivated by real-world problems where data may have unbounded domains. These automata address a demand for a model over large or infinite alphabets, respectively. Both automata models have interesting applications and have been successful in their own right. In this paper, we introduce Symbolic Register Automata, a new model that combines features from both symbolic and register automata, with a view on applications that were previously out of reach. We study their properties and provide algorithms for emptiness, inclusion and equivalence checking, together with experimental results.

1 Introduction

Finite automata are a ubiquitous formalism that is simple enough to model many real-life systems and phenomena. They enjoy a large variety of theoretical properties that in turn play a role in practical applications. For example, finite automata are closed under Boolean operations, and have decidable emptiness and equivalence checking procedures. Unfortunately, finite automata have a fundamental limitation: they can only operate over finite (and typically small) alphabets. Two *orthogonal* families of automata models have been proposed to overcome this: *symbolic automata* and *register automata*. In this paper, we show that these two models can be combined yielding a new powerful model that can cover interesting applications previously out of reach for existing models.

Symbolic finite automata (SFAs) allow transitions to carry predicates over rich first-order alphabet theories, such as linear arithmetic, and therefore extend classic automata to operate over infinite alphabets [12]. For example, an SFA can define the language of all lists of integers in which the first and last elements are positive integer numbers. Despite their increased expressiveness, SFAs enjoy the same closure and decidability properties of finite automata—e.g., closure under Boolean operations and decidable equivalence and emptiness.

This work was partially funded by NSF Grants CCF-1763871, CCF-1750965, a Facebook TAV Research Award, the ERC starting grant Profoundnet (679127) and a Leverhulme Prize (PLP-2016-129). See [10] for the full version of this paper.

© The Author(s) 2019

I. Dillig and S. Tasiran (Eds.): CAV 2019, LNCS 11561, pp. 3–21, 2019.

https://doi.org/10.1007/978-3-030-25540-4_1

Register automata (RA) support infinite alphabets by allowing input characters to be stored in registers during the computation and to be compared against existing values that are already stored in the registers [17]. For example, an RA can define the language of all lists of integers in which all numbers appearing in even positions are the same. RAs do not have some of the properties of finite automata (e.g., they cannot be determinized), but they still enjoy many useful properties that have made them a popular model in static analysis, software verification, and program monitoring [15].

In this paper, we combine the best features of these two models—first order alphabet theories and registers—into a new model, *symbolic register automata* (SRA). SRAs are strictly more expressive than SFAs and RAs. For example, an SRA can define the language of all lists of integers in which the first and last elements are positive rational numbers and all numbers appearing in even positions are the same. This language is not recognizable by either an SFA nor by an RA.

While other attempts at combining symbolic automata and registers have resulted in undecidable models with limited closure properties [11], we show that SRAs enjoy the same closure and decidability properties of (non-symbolic) register automata. We propose a new application enabled by SRAs and implement our model in an open-source automata library.

In summary, our contributions are:

- Symbolic Register Automata (SRA): a new automaton model that can handle complex alphabet theories while allowing symbols at arbitrary positions in the input string to be compared using equality (Sect. 3).
- A thorough study of the properties of SRAs. We show that SRAs are closed under intersection, union and (deterministic) complementation, and provide algorithms for emptiness and forward (bi)simulation (Sect. 4).
- A study of the effectiveness of our SRA implementation on handling regular expressions with back-references (Sect. 5). We compile a set of benchmarks from existing regular expressions with back-references (e.g., $(\backslash d)[a-z]^*\backslash 1$) and show that SRAs are an effective model for such expressions and existing models such as SFAs and RAs are not. Moreover, we show that SRAs are more efficient than the `java.util.regex` library for matching regular expressions with back-references.

2 Motivating Example

In this section, we illustrate the capabilities of symbolic register automata using a simple example. Consider the regular expression r_p shown in Fig. 1a. This expression, given a sequence of product descriptions, checks whether the products have the same code and lot number. The reader might not be familiar with some of the unusual syntax of this expression. In particular, r_p uses two back-references $\backslash 1$ and $\backslash 2$. The semantics of this construct is that the string matched by the regular expression for $\backslash 1$ (resp. $\backslash 2$) should be exactly the string that matched the subregular expression r appearing between the first (resp. second)

$C:(.\{3\}) \ L:(.) \ D:[^\backslash s]+(C:\backslash 1 \ L:\backslash 2 \ D:[^\backslash s]+)+$

(a) Regular expression r_p (with back-reference).

$C:X4a \ L:4 \ D:bottle \ C:X4a \ L:4 \ D:jar$

(b) Example text matched by r_p .

$C:X4a \ L:4 \ D:bottle \ C:X5a \ L:4 \ D:jar$

(c) Example text *not* matched by r_p .

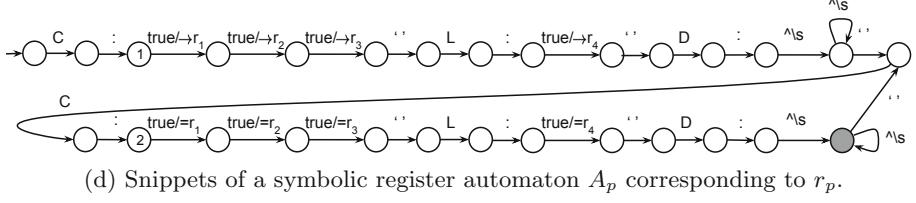


Fig. 1. Regular expression for matching products with same code and lot number—i.e., the characters of C and L are the same in all the products.

two parenthesis, in this case $(.\{3\})$ (resp. $(.)$). Back-references allow regular expressions to check whether the encountered text is the same or is different from a string/character that appeared earlier in the input (see Figs. 1b and c for examples of positive and negative matches).

Representing this complex regular expression using an automaton model requires addressing several challenges. The expression r_p :

1. operates over large input alphabets consisting of upwards of 2^{16} characters;
2. uses complex character classes (e.g., $\backslash s$) to describe different sets of characters in the input;
3. adopts back-references to detect repeated strings in the input.

Existing automata models do not address one or more of these challenges. Finite automata require one transition for each character in the input alphabet and blow-up when representing large alphabets. Symbolic finite automata (SFA) allow transitions to carry predicates over rich structured first-order alphabet theories and can describe, for example, character classes [12]. However, SFAs cannot directly check whether a character or a string is repeated in the input. An SFA for describing the regular expression r_p would have to store the characters after C: directly in the states to later check whether they match the ones of the second product. Hence, the smallest SFA for this example would require billions of states! Register automata (RA) and their variants can store characters in registers during the computation and compare characters against values already stored in the registers [17]. Hence, RAs can check whether the two products have the same code. However, RAs only operate over unstructured infinite alphabets and cannot check, for example, that a character belongs to a given class.

The model we propose in this paper, *symbolic register automata* (SRA), combines the best features of SFAs and RAs—first-order alphabet theories and registers—and can address all the three aforementioned challenges. Figure 1d shows a snippet of a symbolic register automaton A_p corresponding to r_p . Each transition in A_p is labeled with a predicate that describes what characters can

trigger the transition. For example, $\wedge \backslash s$ denotes that the transition can be triggered by any non-space character, L denotes that the transition can be triggered by the character L , and $true$ denotes that the transition can be triggered by any character. Transitions of the form $\varphi / \rightarrow r_i$ denote that, if a character x satisfies the predicate φ , the character is then stored in the register r_i . For example, the transition out of state 1 reads any character and stores it in register r_1 . Finally, transitions of the form $\varphi / = r_i$ are triggered if a character x satisfies the predicate φ and x is the same character as the one stored in r_i . For example, the transition out of state 2 can only be triggered by the same character that was stored in r_1 when reading the transition out state 1—i.e., the first characters in the product codes should be the same.

SRAs are a natural model for describing regular expressions like r_p , where capture groups are of bounded length, and hence correspond to finitely-many registers. The SRA A_p has fewer than 50 states (vs. more than 100 billion for SFAs) and can, for example, be used to check whether an input string matches the given regular expression (e.g., monitoring). More interestingly, in this paper we study the closure and decidability properties of SRAs and provide an implementation for our model. For example, consider the following regular expression r_{pC} that only checks whether the product codes are the same, but not the lot numbers:

$$C:(.\{3\})\ L:.\ D:[\wedge \backslash s]+(C:\backslash 1\ L:.\ D:[\wedge \backslash s]+)+$$

The set of strings accepted by r_{pC} is a superset of the set of strings accepted by r_p . In this paper, we present simulation and bisimulation algorithms that can check this property. Our implementation can show that r_p subsumes r_{pC} in 25 s and we could not find other tools that can prove the same property.

3 Symbolic Register Automata

In this section we introduce some preliminary notions, we define symbolic register automata and a variant that will be useful in proving decidability properties.

Preliminaries. An *effective Boolean algebra* \mathcal{A} is a tuple $(\mathcal{D}, \Psi, [\cdot], \perp, \top, \wedge, \vee, \neg)$, where: \mathcal{D} is a set of domain elements; Ψ is a set of predicates closed under the Boolean connectives and $\perp, \top \in \Psi$. The denotation function $[\cdot]: \Psi \rightarrow 2^{\mathcal{D}}$ is such that $[\perp] = \emptyset$ and $[\top] = \mathcal{D}$, for all $\varphi, \psi \in \Psi$, $[\varphi \vee \psi] = [\varphi] \cup [\psi]$, $[\varphi \wedge \psi] = [\varphi] \cap [\psi]$, and $[\neg \varphi] = \mathcal{D} \setminus [\varphi]$. For $\varphi \in \Psi$, we write $\text{isSat}(\varphi)$ whenever $[\varphi] \neq \emptyset$ and say that φ is *satisfiable*. \mathcal{A} is *decidable* if isSat is decidable. For each $a \in \mathcal{D}$, we assume predicates $\text{atom}(a)$ such that $[\text{atom}(a)] = \{a\}$.

Example 1. The theory of linear integer arithmetic forms an effective BA, where $\mathcal{D} = \mathbb{Z}$ and Ψ contains formulas $\varphi(x)$ in the theory with one fixed integer variable. For example, $\text{div}_k := (x \bmod k) = 0$ denotes the set of all integers divisible by k .

Notation. Given a set S , we write $\mathcal{P}(S)$ for its powerset. Given a function $f: A \rightarrow B$, we write $f[a \mapsto b]$ for the function such that $f[a \mapsto b](a) = b$ and $f[a \mapsto b](x) = f(x)$, for $x \neq a$. Analogously, we write $f[S \mapsto b]$, with $S \subseteq A$, to map multiple values to the same b . The *pre-image* of f is the function $f^{-1}: \mathcal{P}(B) \rightarrow \mathcal{P}(A)$ given by $f^{-1}(S) = \{a \mid \exists b \in S: b = f(a)\}$; for readability, we will write $f^{-1}(x)$ when $S = \{x\}$. Given a relation $\mathcal{R} \subseteq A \times B$, we write aRb for $(a, b) \in \mathcal{R}$.

Model Definition. Symbolic register automata have transitions of the form:

$$p \xrightarrow{\varphi/E,I,U} q$$

where p and q are states, φ is a predicate from a fixed effective Boolean algebra, and E, I, U are subsets of a fixed finite set of registers R . The intended interpretation of the above transition is: an input character a can be read in state q if (i) $a \in \llbracket \varphi \rrbracket$, (ii) the content of all the registers in E is *equal* to a , and (iii) the content of all the registers in I is *different* from a . If the transition succeeds then a is stored into all the registers U and the automaton moves to q .

Example 2. The transition labels in Fig. 1d have been conveniently simplified to ease intuition. These labels correspond to full SRA labels as follows:

$$\varphi/\neg r \implies \varphi/\emptyset, \emptyset, \{r\} \quad \varphi/r \implies \varphi/\{r\}, \emptyset, \emptyset \quad \varphi \implies \varphi/\emptyset, \emptyset, \emptyset .$$

Given a set of registers R , the transitions of an SRA have labels over the following set: $L_R = \Psi \times \{(E, I, U) \in \mathcal{P}(R) \times \mathcal{P}(R) \times \mathcal{P}(R) \mid E \cap I = \emptyset\}$. The condition $E \cap I = \emptyset$ guarantees that register constraints are always satisfiable.

Definition 1 (Symbolic Register Automaton). A symbolic register automaton (SRA) is a 6-tuple $(R, Q, q_0, v_0, F, \Delta)$, where R is a finite set of registers, Q is a finite set of states, $q_0 \in Q$ is the initial state, $v_0: R \rightarrow \mathcal{D} \cup \{\#\}$ is the initial register assignment (if $v_0(r) = \#$, the register r is considered empty), $F \subseteq Q$ is a finite set of final states, and $\Delta \subseteq Q \times L_R \times Q$ is the transition relation. Transitions $(p, (\varphi, \ell), q) \in \Delta$ will be written as $p \xrightarrow{\varphi/\ell} q$.

An SRA can be seen as a finite description of a (possibly infinite) labeled transition system (LTS), where states have been assigned concrete register values, and transitions read a single symbol from the potentially infinite alphabet. This so-called *configuration LTS* will be used in defining the semantics of SRAs.

Definition 2 (Configuration LTS). Given an SRA \mathcal{S} , the configuration LTS $\text{CLTS}(\mathcal{S})$ is defined as follows. A configuration is a pair (p, v) where $p \in Q$ is a state in \mathcal{S} and a $v: R \rightarrow \mathcal{D} \cup \{\#\}$ is register assignment; (q_0, v_0) is called the initial configuration; every (q, v) such that $q \in F$ is a final configuration. The set of transitions between configurations is defined as follows:

$$(p, v) \xrightarrow{a} (q, v[U \mapsto a]) \in \text{CLTS}(\mathcal{S})$$

$$\frac{p \xrightarrow{\varphi/E,I,U} q \in \Delta \quad E \subseteq v^{-1}(a) \quad I \cap v^{-1}(a) = \emptyset}{(p, v) \xrightarrow{a} (q, v[U \mapsto a]) \in \text{CLTS}(\mathcal{S})}$$

Intuitively, the rule says that a SRA transition from p can be instantiated to one from (p, v) that reads a when the registers containing the value a , namely $v^{-1}(a)$, satisfy the constraint described by E, I (a is contained in registers E but not in I). If the constraint is satisfied, all registers in U are assigned a .

A *run* of the SRA \mathcal{S} is a sequence of transitions in $\text{CLTS}(\mathcal{S})$ starting from the initial configuration. A configuration is *reachable* whenever there is a run ending up in that configuration. The *language* of an SRA \mathcal{S} is defined as

$$\mathcal{L}(\mathcal{S}) := \{a_1 \dots a_n \in \mathcal{D}^n \mid \exists (q_0, v_0) \xrightarrow{a_1} \dots \xrightarrow{a_n} (q_n, v_n) \in \text{CLTS}(\mathcal{S}), q_n \in F\}$$

An SRA \mathcal{S} is *deterministic* if its configuration LTS is; namely, for every word $w \in \mathcal{D}^*$ there is at most one run in $\text{CLTS}(\mathcal{S})$ spelling w . Determinism is important for some application contexts, e.g., for runtime monitoring. Since SRAs subsume RAs, nondeterministic SRAs are strictly more expressive than deterministic ones, and language equivalence is undecidable for nondeterministic SRAs [27].

We now introduce the notions of *simulation* and *bisimulation* for SRAs, which capture whether one SRA behaves “at least as” or “exactly as” another one.

Definition 3 ((Bi)simulation for SRAs). A simulation \mathcal{R} on SRAs \mathcal{S}_1 and \mathcal{S}_2 is a binary relation \mathcal{R} on configurations such that $(p_1, v_1)\mathcal{R}(p_2, v_2)$ implies:

- if $p_1 \in F_1$ then $p_2 \in F_2$;
- for each transition $(p_1, v_1) \xrightarrow{a} (q_1, w_1)$ in $\text{CLTS}(\mathcal{S}_1)$, there exists a transition $(p_2, v_2) \xrightarrow{a} (q_2, w_2)$ in $\text{CLTS}(\mathcal{S}_2)$ such that $(q_1, w_1)\mathcal{R}(q_2, w_2)$.

A simulation \mathcal{R} is a bisimulation if \mathcal{R}^{-1} is also a simulation. We write $\mathcal{S}_1 \prec \mathcal{S}_2$ (resp. $\mathcal{S}_1 \sim \mathcal{S}_2$) whenever there is a simulation (resp. bisimulation) \mathcal{R} such that $(q_{01}, v_{01})\mathcal{R}(q_{02}, v_{02})$, where (q_{0i}, v_{0i}) is the initial configuration of \mathcal{S}_i , for $i = 1, 2$.

We say that an SRA is *complete* whenever for every configuration (p, v) and $a \in \mathcal{D}$ there is a transition $(p, v) \xrightarrow{a} (q, w)$ in $\text{CLTS}(\mathcal{S})$. The following results connect similarity and language inclusion.

Proposition 1. If $\mathcal{S}_1 \prec \mathcal{S}_2$ then $\mathcal{L}(\mathcal{S}_1) \subseteq \mathcal{L}(\mathcal{S}_2)$. If \mathcal{S}_1 and \mathcal{S}_2 are deterministic and complete, then the other direction also holds.

It is worth noting that given a deterministic SRA we can define its *completion* by adding transitions so that every value $a \in \mathcal{D}$ can be read from any state.

Remark 1. RAs and SFAs can be encoded as SRAs on the same state-space:

- An RA is encoded as an SRA with all transition guards \top ;
- an SFA can be encoded as an SRA with $R = \emptyset$, with each SFA transition $p \xrightarrow{\varphi} q$ encoded as $p \xrightarrow{\varphi/\emptyset,\emptyset,\emptyset} q$. Note that the absence of registers implies that the CLTS always has finitely many configurations.

SRAs are *strictly more expressive* than both RAs and SFAs. For instance, the language $\{n_0 n_1 \dots n_k \mid n_0 = n_k, \text{even}(n_i), n_i \in \mathbb{Z}, i = 1, \dots, k\}$ of finite sequences of even integers where the first and last one coincide, can be recognized by an SRA, but not by an RA or by an SFA.

Boolean Closure Properties. SRAs are closed under intersection and union. Intersection is given by a standard product construction whereas union is obtained by adding a new initial state that mimics the initial states of both automata.

Proposition 2 (Closure under intersection and union). *Given SRAs \mathcal{S}_1 and \mathcal{S}_2 , there are SRAs $\mathcal{S}_1 \cap \mathcal{S}_2$ and $\mathcal{S}_1 \cup \mathcal{S}_2$ such that $\mathcal{L}(\mathcal{S}_1 \cap \mathcal{S}_2) = \mathcal{L}(\mathcal{S}_1) \cap \mathcal{L}(\mathcal{S}_2)$ and $\mathcal{L}(\mathcal{S}_1 \cup \mathcal{S}_2) = \mathcal{L}(\mathcal{S}_1) \cup \mathcal{L}(\mathcal{S}_2)$.*

SRAs in general are not closed under complementation, because RAs are not. However, we still have closure under complementation for a subclass of SRAs.

Proposition 3. *Let \mathcal{S} be a complete and deterministic SRA, and let $\bar{\mathcal{S}}$ be the SRA defined as \mathcal{S} , except that its final states are $Q \setminus F$. Then $\mathcal{L}(\bar{\mathcal{S}}) = \mathcal{D}^* \setminus \mathcal{L}(\mathcal{S})$.*

4 Decidability Properties

In this section we will provide algorithms for checking determinism and emptiness for an SRA, and (bi)similarity of two SRAs. Our algorithms leverage *symbolic* techniques that use the finite syntax of SRAs to indirectly operate over the underlying configuration LTS, which can be infinite.

Single-Valued Variant. To study decidability, it is convenient to restrict register assignments to *injective* ones on non-empty registers, that is functions $v: R \rightarrow \mathcal{D} \cup \{\#\}$ such that $v(r) = v(s)$ and $v(r) \neq \#$ implies $r = s$. This is also the approach taken for RAs in the seminal papers [17, 27]. Both for RAs and SRAs, this restriction does not affect expressivity. We say that an SRA is *single-valued* if its initial assignment v_0 is injective on non-empty registers. For single-valued SRAs, we only allow two kinds of transitions:

Read transition: $p \xrightarrow{\varphi/r= q}$ triggers when $a \in \llbracket \varphi \rrbracket$ and a is already stored in r .

Fresh transition: $p \xrightarrow{\varphi/r^\bullet q}$ triggers when the input $a \in \llbracket \varphi \rrbracket$ and a is *fresh*, i.e., is not stored in any register. After the transition, a is stored into r .

SRAs and their single-valued variants have the same expressive power. Translating single-valued SRAs to ordinary ones is straightforward:

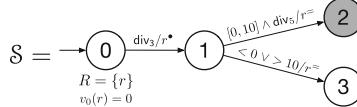
$$p \xrightarrow{\varphi/r= q} \implies p \xrightarrow{\varphi/\{r\}, \emptyset, \emptyset q} \quad p \xrightarrow{\varphi/r^\bullet q} \implies p \xrightarrow{\varphi/\emptyset, R, \{r\} q}$$

The opposite translation requires a state-space blow up, because we need to encode register equalities in the states.

Theorem 1. *Given an SRA \mathcal{S} with n states and r registers, there is a single-valued SRA \mathcal{S}' with $\mathcal{O}(nr^r)$ states and $r+1$ registers such that $\mathcal{S} \sim \mathcal{S}'$. Moreover, the translation preserves determinism.*

Normalization. While our techniques are inspired by analogous ones for non-symbolic RAs, SRAs present an additional challenge: they can have arbitrary predicates on transitions. Hence, the values that each transition can read, and thus which configurations it can reach, depend on the history of past transitions and their predicates. This problem emerges when checking reachability and similarity, because a transition may be *disabled* by particular register values, and so lead to unsound conclusions, a problem that does not exist in register automata.

Example 3. Consider the SRA below, defined over the BA of integers.



All predicates on transitions are satisfiable, yet $\mathcal{L}(\mathcal{S}) = \emptyset$. To go from 0 to 1, \mathcal{S} must read a value n such that $\text{div}_3(n)$ and $n \neq 0$ and then n is stored into r . The transition from 1 to 2 can only happen if the content of r also satisfies $\text{div}_5(n)$ and $n \in [0, 10]$. However, there is no n satisfying $\text{div}_3(n) \wedge n \neq 0 \wedge \text{div}_5(n) \wedge n \in [0, 10]$, hence the transition from 1 to 2 never happens.

To handle the complexity caused by predicates, we introduce a way of *normalizing* an SRA to an equivalent one that *stores additional information about input predicates*. We first introduce some notation and terminology.

A register abstraction θ for \mathcal{S} , used to “keep track” of the domain of registers, is a family of predicates indexed by the registers R of \mathcal{S} . Given a register assignment v , we write $v \models \theta$ whenever $v(r) \in [\theta_r]$ for $v(r) \neq \sharp$, and $\theta_r = \perp$ otherwise. Hereafter we shall only consider “meaningful” register abstractions, for which there is at least one assignment v such that $v \models \theta$.

With the contextual information about register domains given by θ , we say that a transition $p \xrightarrow{\varphi/\ell} q \in \Delta$ is *enabled by θ* whenever it has at least an instance $(p, v) \xrightarrow{a} (q, w)$ in $\text{CLTS}(\mathcal{S})$, for all $v \models \theta$. Enabled transitions are important when reasoning about reachability and similarity.

Checking whether a transition has at least one realizable instance in the CLTS is difficult in practice, especially when $\ell = r^\bullet$, because it amounts to checking whether $[\varphi] \setminus \text{img}(v) \neq \emptyset$, for all injective $v \models \theta$.

To make the check for enabledness practical we will use minterms. For a set of predicates Φ , a *minterm* is a minimal satisfiable Boolean combination of all predicates that occur in Φ . Minterms are the analogue of atoms in a complete atomic Boolean algebra. E.g. the set of predicates $\Phi = \{x > 2, x < 5\}$ over the theory of linear integer arithmetic has minterms $\text{mint}(\Phi) = \{x > 2 \wedge x < 5, \neg x > 2 \wedge x < 5, x > 2 \wedge \neg x < 5\}$. Given $\psi \in \text{mint}(\Phi)$ and $\varphi \in \Phi$, we will write $\varphi \sqsubset \psi$ whenever φ appears non-negated in ψ , for instance $(x > 2) \sqsubset (x > 2 \wedge \neg x < 5)$. A crucial property of minterms is that they do not overlap, i.e., $\text{isSat}(\psi_1 \wedge \psi_2)$ if and only if $\psi_1 = \psi_2$, for ψ_1 and ψ_2 minterms.

Lemma 1 (Enabledness). Let θ be a register abstraction such that θ_r is a minterm, for all $r \in R$. If φ is a minterm, then $p \xrightarrow{\varphi/\ell} q$ is enabled by θ iff:

(1) if $\ell = r^=$, then $\varphi = \theta_r$; (2) if $\ell = r^\bullet$, then $|[\![\varphi]\!]| > \mathcal{E}(\theta, \varphi)$,
 where $\mathcal{E}(\theta, \varphi) = |\{r \in R \mid \theta_r = \varphi\}|$ is the # of registers with values from $[\![\varphi]\!]$.

Intuitively, (1) says that if the transition reads a symbol stored in r satisfying φ , the symbol must also satisfy θ_r , the range of r . Because φ and θ_r are minterms, this only happens when $\varphi = \theta_r$. (2) says that the enabling condition $[\![\varphi]\!] \setminus \text{img}(v) \neq \emptyset$, for all injective $v \models \theta$, holds if and only if there are fewer registers storing values from φ than the cardinality of φ . That implies we can always find a fresh element in $[\![\varphi]\!]$ to enable the transition. Registers holding values from φ are exactly those $r \in R$ such that $\theta_r = \varphi$. Both conditions can be effectively checked: the first one is a simple predicate-equivalence check, while the second one amounts to checking whether φ holds for at least a certain number k of distinct elements. This can be achieved by checking satisfiability of $\varphi \wedge \neg \text{atom}(a_1) \wedge \dots \wedge \neg \text{atom}(a_{k-1})$, for a_1, \dots, a_{k-1} distinct elements of $[\![\varphi]\!]$.

Remark 2. Using single-valued SRAs to check enabledness might seem like a restriction. However, if one would start from a generic SRA, the process to check enabledness would contain an extra step: for each state p , we would have to keep track of all possible equations among registers. In fact, register equalities determine whether (i) register constraints of an outgoing transition are satisfiable; (ii) how many elements of the guard we need for the transition to happen, analogously to condition 2 of Lemma 1. Generating such equations is the key idea behind Theorem 1, and corresponds precisely to turning the SRA into a single-valued one.

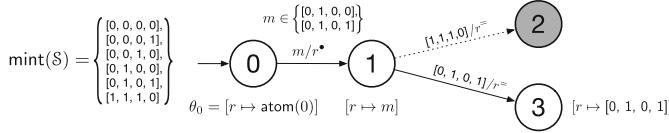
Given any SRA, we can use the notion of register abstraction to build an equivalent *normalized* SRA, where (i) states keep track of how the domains of registers change along transitions, (ii) transitions are obtained by breaking the one of the original SRA into minterms and discarding the ones that are disabled according to Lemma 1. In the following we write $\text{mint}(\mathcal{S})$ for the minterms for the set of predicates $\{\varphi \mid p \xrightarrow{\varphi/\ell} q \in \Delta\} \cup \{\text{atom}(v_0(r)) \mid v_0(r) \in \mathcal{D}, r \in R\}$. Observe that an atomic predicate always has an equivalent minterm, hence we will use atomic predicates to define the initial register abstraction.

Definition 4 (Normalized SRA). Given an SRA \mathcal{S} , its normalization $\mathbf{N}(\mathcal{S})$ is the SRA $(R, \mathbf{N}(Q), \mathbf{N}(q_0), v_0, \mathbf{N}(F), \mathbf{N}(\Delta))$ where:

- $\mathbf{N}(Q) = \{\theta \mid \theta \text{ is a register abstraction over } \text{mint}(\mathcal{S}) \cup \{\perp\}\} \times Q$; we will write $\theta \triangleright q$ for $(\theta, q) \in \mathbf{N}(Q)$.
- $\mathbf{N}(q_0) = \theta_0 \triangleright q_0$, where $(\theta_0)_r = \text{atom}(v_0(r))$ if $v_0(r) \in \mathcal{D}$, and $(\theta_0)_r = \perp$ if $v_0(r) = \sharp$;
- $\mathbf{N}(F) = \{\theta \triangleright p \in \mathbf{N}(Q) \mid p \in F\}$
- $\mathbf{N}(\Delta) = \{\theta \triangleright p \xrightarrow{\theta_r/r^=} \theta \triangleright q \mid p \xrightarrow{\varphi/r^=} q \in \Delta, \varphi \sqsubset \theta_r\} \cup$
 $\{\theta \triangleright p \xrightarrow{\psi/r^\bullet} \theta[r \mapsto \psi] \triangleright q \mid p \xrightarrow{\varphi/r^\bullet} q \in \Delta, \varphi \sqsubset \psi, |[\![\psi]\!]| > \mathcal{E}(\theta, \psi)\}$

The automaton $N(\mathcal{S})$ enjoys the desired property: each transition from $\theta \triangleright p$ is enabled by θ , by construction. $N(\mathcal{S})$ is always *finite*. In fact, suppose \mathcal{S} has n states, m transitions and r registers. Then $N(\mathcal{S})$ has at most m predicates, and $|\text{mint}(\mathcal{S})|$ is $\mathcal{O}(2^m)$. Since the possible register abstractions are $\mathcal{O}(r2^m)$, $N(\mathcal{S})$ has $\mathcal{O}(nr2^m)$ states and $\mathcal{O}(mr^22^{3m})$ transitions.

Example 4. We now show the normalized version of Example 3. The first step is computing the set $\text{mint}(\mathcal{S})$ of minterms for \mathcal{S} , i.e., the satisfiable Boolean combinations of $\{\text{atom}(0), \text{div}_3, [0, 10] \wedge \text{div}_5, <0\vee> 10\}$. For simplicity, we represent minterms as bitvectors where a 0 component means that the corresponding predicate is negated, e.g., $[1, 1, 1, 0]$ stands for the minterm $\text{atom}(0) \wedge ([0, 10] \wedge \text{div}_3) \wedge \text{div}_5 \wedge \neg(<0\vee> 10)$. Minterms and the resulting SRA $N(\mathcal{S})$ are shown below.



On each transition we show how it is broken down to minterms, and for each state we show the register abstraction (note that state 1 becomes two states in $N(\mathcal{S})$). The transition from 1 to 2 is *not* part of $N(\mathcal{S})$ – this is why it is dotted. In fact, in every register abstraction $[r \mapsto m]$ reachable at state 1, the component for the transition guard $[0, 10] \wedge \text{div}_5$ in the minterm m (3rd component) is 0, i.e., $([0, 10] \wedge \text{div}_5) \not\subseteq m$. Intuitively, this means that r will never be assigned a value that satisfies $[0, 10] \wedge \text{div}_5$. As a consequence, the construction of Definition 4 will not add a transition from 1 to 2.

Finally, we show that the normalized SRA behaves exactly as the original one.

Proposition 4. $(p, v) \sim (\theta \triangleright p, v)$, for all $p \in Q$ and $v \models \theta$. Hence, $\mathcal{S} \sim N(\mathcal{S})$.

Emptiness and Determinism. The transitions of $N(\mathcal{S})$ are always enabled by construction, therefore every path in $N(\mathcal{S})$ always corresponds to a run in $\text{CLTS}(N(\mathcal{S}))$.

Lemma 2. *The state $\theta \triangleright p$ is reachable in $N(\mathcal{S})$ if and only if there is a reachable configuration $(\theta \triangleright p, v)$ in $\text{CLTS}(N(\mathcal{S}))$ such that $v \models \theta$. Moreover, if $(\theta \triangleright p, v)$ is reachable, then all configurations $(\theta \triangleright p, w)$ such that $w \models \theta$ are reachable.*

Therefore, using Proposition 4, we can reduce the reachability and emptiness problems of \mathcal{S} to that of $N(\mathcal{S})$.

Theorem 2 (Emptiness). *There is an algorithm to decide reachability of any configuration of \mathcal{S} , hence whether $\mathcal{L}(\mathcal{S}) = \emptyset$.*

Proof. Let (p, v) be a configuration of \mathcal{S} . To decide whether it is reachable in $\text{CLTS}(\mathcal{S})$, we can perform a visit of $N(\mathcal{S})$ from its initial state, stopping when a

state $\theta \triangleright p$ such that $v \models \theta$ is reached. If we are just looking for a final state, we can stop at any state such that $p \in F$. In fact, by Proposition 4, there is a run in $\text{CLTS}(\mathcal{S})$ ending in (p, v) if and only if there is a run in $\text{CLTS}(\mathbf{N}(\mathcal{S}))$ ending in $(\theta \triangleright p, v)$ such that $v \models \theta$. By Lemma 2, the latter holds if and only if there is a path in $\mathbf{N}(\mathcal{S})$ ending in $\theta \triangleright p$. This algorithm has the complexity of a standard visit of $\mathbf{N}(\mathcal{S})$, namely $\mathcal{O}(nr2^m + mr^22^{3m})$. \square

Now that we characterized which transitions are reachable, we define what it means for a normalized SRA to be deterministic and we show that determinism is preserved by the translation from SRA.

Proposition 5 (Determinism). $\mathbf{N}(\mathcal{S})$ is deterministic if and only if for all reachable transitions $p \xrightarrow{\varphi_1/\ell_1} q_1, p \xrightarrow{\varphi_2/\ell_2} q_2 \in \mathbf{N}(\Delta)$ the following holds: $\varphi_1 \neq \varphi_2$ whenever either (1) $\ell_1 = \ell_2$ and $q_1 \neq q_2$, or; (2) $\ell_1 = r^\bullet$, $\ell_2 = s^\bullet$, and $r \neq s$;

One can check determinism of an SRA by looking at its normalized version.

Proposition 6. \mathcal{S} is deterministic if and only if $\mathbf{N}(\mathcal{S})$ is deterministic.

Similarity and Bisimilarity. We now introduce a symbolic technique to decide similarity and bisimilarity of SRAs. The basic idea is similar to *symbolic (bi)simulation* [20, 27] for RAs. Recall that RAs are SRAs whose transition guards are all \top . Given two RAs \mathcal{S}_1 and \mathcal{S}_2 a symbolic simulation between them is defined over their state spaces Q_1 and Q_2 , not on their configurations. For this to work, one needs to add an extra piece of information about how registers of the two states are related. More precisely, a symbolic simulation is a relation on triples (p_1, p_2, σ) , where $p_1 \in Q_1, p_2 \in Q_2$ and $\sigma \subseteq R_1 \times R_2$ is a *partial injective* function. This function encodes constraints between registers: $(r, s) \in \sigma$ is an equality constraint between $r \in R_1$ and $s \in R_2$, and $(r, s) \notin \sigma$ is an inequality constraint. Intuitively, (p_1, p_2, σ) says that all configurations (p_1, v_1) and (p_2, v_2) such that v_1 and v_2 satisfy σ – e.g., $v_1(r) = v_2(s)$ whenever $(r, s) \in \sigma$ – are in the simulation relation $(p_1, v_1) \prec (p_2, v_2)$. In the following we will use $v_1 \bowtie v_2$ to denote the function encoding constraints among v_1 and v_2 , explicitly: $\sigma(r) = s$ if and only if $v_1(r) = v_2(s)$ and $v_1(r) \neq \sharp$.

Definition 5 (Symbolic (bi)similarity [27]). A symbolic simulation is a relation $\mathcal{R} \subseteq Q_1 \times Q_2 \times \mathcal{P}(R_1 \times R_2)$ such that if $(p_1, p_2, \sigma) \in \mathcal{R}$, then $p_1 \in F_1$ implies $p_2 \in F_2$, and if $p_1 \xrightarrow{\ell} q_1 \in \Delta_1$ ¹ then:

1. if $\ell = r^=$:

- (a) if $r \in \text{dom}(\sigma)$, then there is $p_2 \xrightarrow{\sigma(r)^=} q_2 \in \Delta_2$ such that $(q_1, q_2, \sigma) \in \mathcal{R}$.
- (b) if $r \notin \text{dom}(\sigma)$ then there is $p_2 \xrightarrow{s^\bullet} q_2 \in \Delta_2$ s.t. $(q_1, q_2, \sigma[r \mapsto s]) \in \mathcal{R}$.

¹ We will keep the \top guard implicit for succinctness.

2 if $\ell = r^\bullet$:

- (a) for all $s \in R_2 \setminus \text{img}(\sigma)$, there is $p_2 \xrightarrow{s^=} q_2 \in \Delta_2$ such that $(q_1, q_2, \sigma[r \mapsto s]) \in \mathcal{R}$, and;
- (b) there is $p_2 \xrightarrow{s^\bullet} q_2 \in \Delta_2$ such that $(q_1, q_2, \sigma[r \mapsto s]) \in \mathcal{R}$.

Here $\sigma[r \mapsto s]$ stands for $\sigma \setminus (\sigma^{-1}(s), s) \cup (r, s)$, which ensures that σ stays injective when updated.

Given a symbolic simulation \mathcal{R} , its inverse is defined as $\mathcal{R}^{-1} = \{t^{-1} \mid t \in \mathcal{R}\}$, where $(p_1, p_2, \sigma)^{-1} = (p_2, p_1, \sigma^{-1})$. A symbolic bisimulation \mathcal{R} is a relation such that both \mathcal{R} and \mathcal{R}^{-1} are symbolic simulations.

Case 1 deals with cases when p_1 can perform a transition that reads the register r . If $r \in \text{dom}(\sigma)$, meaning that r and $\sigma(r) \in R_2$ contain the same value, then p_2 must be able to read $\sigma(r)$ as well. If $r \notin \text{dom}(\sigma)$, then the content of r is fresh w.r.t. p_2 , so p_2 must be able to read any fresh value—in particular the content of r . Case 2 deals with the cases when p_1 reads a fresh value. It ensures that p_2 is able to read all possible values that are fresh for p_1 , be them already in some register s —i.e., $s \in R_2 \setminus \text{img}(\sigma)$, case 2(a)—or fresh for p_2 as well—case 2(b). In all these cases, σ must be updated to reflect the new equalities among registers.

Keeping track of equalities among registers is enough for RAs, because the actual content of registers does not determine the capability of a transition to fire (RA transitions have implicit \top guards). As seen in Example 3, this is no longer the case for SRAs: a transition may or may not happen depending on the register assignment being compatible with the transition guard.

As in the case of reachability, normalized SRAs provide the solution to this problem. We will reduce the problem of checking (bi)similarity of \mathcal{S}_1 and \mathcal{S}_2 to that of checking symbolic (bi)similarity on $N(\mathcal{S}_1)$ and $N(\mathcal{S}_2)$, with minor modifications to the definition. To do this, we need to assume that minterms for both $N(\mathcal{S}_1)$ and $N(\mathcal{S}_2)$ are computed over the union of predicates of \mathcal{S}_1 and \mathcal{S}_2 .

Definition 6 (N-simulation). A N-simulation on \mathcal{S}_1 and \mathcal{S}_2 is a relation $\mathcal{R} \subseteq N(Q_1) \times N(Q_2) \times \mathcal{P}(R_1 \times R_2)$, defined as in Definition 5, with the following modifications:

- (i) we require that $\theta_1 \triangleright p_1 \xrightarrow{\varphi_1/\ell_1} \theta'_1 \triangleright q_1 \in N(\Delta_1)$ must be matched by transitions $\theta_2 \triangleright p_2 \xrightarrow{\varphi_2/\ell_2} \theta'_2 \triangleright q_2 \in N(\Delta_2)$ such that $\varphi_2 = \varphi_1$.
- (ii) we modify case 2 as follows (changes are underlined):
 - 2(a)' for all $s \in R_2 \setminus \text{img}(\sigma)$ such that $\varphi_1 = (\theta_2)_s$, there is $\theta_2 \triangleright p_2 \xrightarrow{\varphi_1/s^=} \theta'_2 \triangleright q_2 \in N(\Delta_2)$ such that $(\theta'_1 \triangleright q_1, \theta'_2 \triangleright q_2, \sigma[r \mapsto s]) \in \mathcal{R}$, and;
 - 2(b)' if $\mathcal{E}(\theta_1, \varphi_1) + \mathcal{E}(\theta_2, \varphi_1) < |\llbracket \varphi_1 \rrbracket|$, then there is $\theta_2 \triangleright p_2 \xrightarrow{\varphi_1/s^\bullet} \theta'_2 \triangleright q_2 \in N(\Delta_2)$ such that $(\theta'_1 \triangleright q_1, \theta'_2 \triangleright q_2, \sigma[r \mapsto s]) \in \mathcal{R}$.

A N-bisimulation \mathcal{R} is a relation such that both \mathcal{R} and \mathcal{R}^{-1} are N-simulations.

We write $\mathcal{S}_1 \xrightarrow{N} \mathcal{S}_2$ (resp. $\mathcal{S}_1 \xrightarrow{N} \mathcal{S}_2$) if there is a N-simulation (resp. bisimulation) \mathcal{R} such that $(N(q_{01}), N(q_{02}), v_{01} \bowtie v_{02}) \in \mathcal{R}$.

The intuition behind this definition is as follows. Recall that, in a normalized SRA, transitions are defined over minterms, which cannot be further broken down, and are mutually disjoint. Therefore two transitions can read the same values if and only if they have the same minterm guard. Thus condition (i) makes sure that matching transitions can read exactly the same set of values. Analogously, condition (ii) restricts how a fresh transition of $N(S_1)$ must be matched by one of $N(S_2)$: 2(a)' only considers transitions of $N(S_2)$ reading registers $s \in R_2$ such that $\varphi_1 = (\theta_2)_s$ because, by definition of normalized SRA, $\theta_2 \triangleright p_2$ has no such transition if this condition is not met. Condition 2(b)' amounts to requiring a fresh transition of $N(S_2)$ that is enabled by both θ_1 and θ_2 (see Lemma 1), i.e., that can read a symbol that is fresh w.r.t. both $N(S_1)$ and $N(S_2)$.

N -simulation is sound and complete for standard simulation.

Theorem 3. $S_1 \prec S_2$ if and only if $S_1 \stackrel{N}{\prec} S_2$.

As a consequence, we can decide similarity of SRAs via their normalized versions. N -simulation is a relation over a finite set, namely $N(Q_1) \times N(Q_2) \times \mathcal{P}(R_1 \times R_2)$, therefore N -similarity can always be decided in finite time. We can leverage this result to provide algorithms for checking language inclusion/equivalence for deterministic SRAs (recall that they are undecidable for non-deterministic ones).

Theorem 4. Given two deterministic SRAs S_1 and S_2 , there are algorithms to decide $\mathcal{L}(S_1) \subseteq \mathcal{L}(S_2)$ and $\mathcal{L}(S_1) = \mathcal{L}(S_2)$.

Proof. By Proposition 1 and Theorem 3, we can decide $\mathcal{L}(S_1) \subseteq \mathcal{L}(S_2)$ by checking $S_1 \stackrel{N}{\prec} S_2$. This can be done algorithmically by iteratively building a relation \mathcal{R} on triples that is an N -simulation on $N(S_1)$ and $N(S_2)$. The algorithm initializes \mathcal{R} with $(N(q_{01}), N(q_{02}), v_{01} \bowtie v_{02})$, as this is required to be in \mathcal{R} by Definition 6. Each iteration considers a candidate triple t and checks the conditions for N -simulation. If satisfied, it adds t to \mathcal{R} and computes the next set of candidate triples, i.e., those which are required to belong to the simulation relation, and adds them to the list of triples still to be processed. If not, the algorithm returns $\mathcal{L}(S_1) \not\subseteq \mathcal{L}(S_2)$. The algorithm terminates returning $\mathcal{L}(S_1) \subseteq \mathcal{L}(S_2)$ when no triples are left to process. Determinism of S_1 and S_2 , and hence of $N(S_1)$ and $N(S_2)$ (by Proposition 6), ensures that computing candidate triples is deterministic. To decide $\mathcal{L}(S_1) = \mathcal{L}(S_2)$, at each iteration we need to check that both t and t^{-1} satisfy the conditions for N -simulation.

If S_1 and S_2 have, respectively, n_1, n_2 states, m_1, m_2 transitions, and r_1, r_2 registers, the normalized versions have $\mathcal{O}(n_1 r_1 2^{m_1})$ and $\mathcal{O}(n_2 r_2 2^{m_2})$ states. Each triple, taken from the finite set $N(Q_1) \times N(Q_2) \times \mathcal{P}(R_1 \times R_2)$, is processed exactly once, so the algorithm iterates $\mathcal{O}(n_1 n_2 r_1 r_2 2^{m_1+m_2+r_1 r_2})$ times. \square

5 Evaluation

We have implemented SRAs in the open-source Java library SVPALib [26]. In our implementation, constructions are computed lazily when possible (e.g., the

normalized SRA for emptiness and (bi)similarity checks). All experiments were performed on a machine with 3.5 GHz Intel Core i7 CPU with 16 GB of RAM (JVM 8 GB), with a timeout value of 300 s. The goal of our evaluation is to answer the following research questions:

- Q1:** Are SRAs more succinct than existing models when processing strings over large but finite alphabets? (Sect. 5.1)
- Q2:** What is the performance of membership for deterministic SRAs and how does it compare to the matching algorithm in `java.util.regex`? (Sect. 5.2)
- Q3:** Are SRA decision procedures practical? (Sect. 5.3)

Benchmarks. We focus on regular expressions with back-references, therefore all our benchmarks operate over the Boolean algebra of Unicode characters with interval—i.e., the set of characters is the set of all 2^{16} UTF-16 characters and the predicates are union of intervals (e.g., `[a-zA-Z]`).² Our benchmark set contains 19 SRAs that represent variants of regular expressions with back-references obtained from the regular-expression crowd-sourcing website RegExLib [23]. The expressions check whether inputs have, for example, matching first/last name initials or both (Name-F, Name-L and Name), correct Product Codes/Lot number of total length n (Pr-C_n , Pr-CL_n), matching XML tags (XML), and IP addresses that match for n positions (IP_n). We also create variants of the product benchmark presented in Sect. 2 where we vary the numbers of characters in the code and lot number. All the SRAs are deterministic.

5.1 Succinctness of SRAs vs SFAs

In this experiment, we relate the size of SRAs over finite alphabets to the size of the smallest equivalent SFAs. For each SRA, we construct the equivalent SFA by equipping the state space with the values stored in the registers at each step (this construction effectively builds the configuration LTS). Figure 2a shows the results. As expected, SFAs tend to blow up in size when the SRA contains multiple registers or complex register values. In cases where the register values range over small sets (e.g., `[0-9]`) it is often feasible to build an SFA equivalent to the SRA, but the construction always yields very large automata. In cases where the registers can assume many values (e.g., 2^{16}) SFAs become prohibitively large and do not fit in memory. To answer **Q1**, even for finite alphabets, **it is not feasible to compile SRAs to SFAs**. Hence, SRAs are a succinct model.

5.2 Performance of Membership Checking

In this experiment, we measure the performance of SRA membership, and we compare it with the performance of the `java.util.regex` matching algorithm.

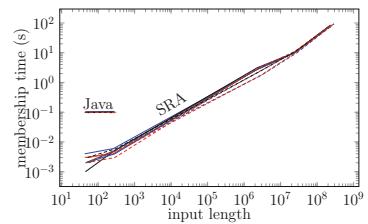
² Our experiments are over finite alphabets, but the Boolean algebra can be infinite by taking the alphabet to be positive integers and allowing intervals to contain ∞ as upper bound. This modification does not affect the running time of our procedures, therefore we do not report it.

	SRA				SFA	
	states	tr	reg	reg	states	tr
IP2	44	46	3	10	4,013	4,312
IP3	44	46	4	10	39,113	42,112
IP4	44	46	5	10	372,113	402,112
IP6	44	46	7	10	—	—
IP9	44	46	10	10	—	—
Name-F	7	10	2	26	201	300
Name-L	7	10	2	26	129	180
Name	7	10	3	26	3,201	4,500
XML	12	16	4	52	—	—
Pr-C2	26	28	3	2^{16}	—	—
Pr-C3	28	30	4	2^{16}	—	—
Pr-C4	30	32	5	2^{16}	—	—
Pr-C6	34	36	7	2^{16}	—	—
Pr-C9	40	42	10	2^{16}	—	—
Pr-CL2	26	28	3	2^{16}	—	—
Pr-CL3	28	30	4	2^{16}	—	—
Pr-CL4	30	32	5	2^{16}	—	—
Pr-CL6	34	36	7	2^{16}	—	—
Pr-CL9	40	42	10	2^{16}	—	—

(a) Size of SRAs vs SFAs. (—) denotes the SFA didn't fit in memory. |reg| denotes how many different characters a register stored.

SRA	\mathcal{S}_1	SRA \mathcal{S}_2	$\mathcal{L}_1 = \emptyset$	$\mathcal{L}_1 = \mathcal{L}_2$	$\mathcal{L}_2 \subseteq \mathcal{L}_1$
Pr-C2	Pr-CL2	Pr-CL2	0.125s	0.905s	3.426s
Pr-C3	Pr-CL3	Pr-CL3	1.294s	5.558s	24.688s
Pr-C4	Pr-CL4	Pr-CL4	13.577s	55.595s	—
Pr-C6	Pr-CL6	Pr-CL6	—	—	—
Pr-CL2	Pr-C2	Pr-C2	1.067s	0.952s	0.889s
Pr-CL3	Pr-C3	Pr-C3	10.998s	11.104s	11.811s
Pr-CL4	Pr-C4	Pr-C4	—	—	—
Pr-CL6	Pr-C6	Pr-C6	—	—	—
IP-2	IP-3	IP-3	0.125s	0.408s	1.845s
IP-3	IP-4	IP-4	1.288s	2.953s	21.627s
IP-4	IP-6	IP-6	18.440s	42.727s	—
IP-6	IP-9	IP-9	—	—	—

(b) Performance of decision procedures.
In the table $\mathcal{L}_i = \mathcal{L}(\mathcal{S}_i)$, for $i = 1, 2$.



(c) SRA membership and Java `regex` matching performance. Missing data points for Java are stack overflows.

Fig. 2. Experimental results.

For each benchmark, we generate inputs of length varying between approximately 100 and 10^8 characters and measure the time taken to check membership. Figure 2c shows the results. The performance of SRAs (resp. Java) is not particularly affected by the size of the expression. Hence, the lines for different expressions mostly overlap. As expected, for SRAs the time taken to check membership grows linearly in the size of the input (axes are log scale). Remarkably, even though our implementation does not employ particular input processing optimizations, it can still check membership for strings with tens of millions of characters in less than 10s. We have found that our implementation is more efficient than the Java `regex` library, matching the same input an average of 50 times faster than `java.util.regex.Matcher`. `java.util.regex.Matcher` seems to make use of a recursive algorithm to match back-references, which means it does not scale well. Even when given the maximum stack size, the JVM will return a Stack Overflow for inputs as small as 20,000 characters. Our implementation can match such strings in less than 2s. To answer **Q2**, **deterministic SRAs can be efficiently executed on large inputs and perform better than the `java.util.regex` matching algorithm.**

5.3 Performance of Decision Procedures

In this experiment, we measure the performance of SRAs simulation and bisimulation algorithms. Since all our SRAs are deterministic, these two checks correspond to language equivalence and inclusion. We select pairs of benchmarks for which the above tests are meaningful (e.g., variants of the problem discussed at the end of Sect. 2). The results are shown in Fig. 2b. As expected, due to the translation to single-valued SRAs, our decision procedures do not scale well in the number of registers. This is already the case for classic register automata and it is not a surprising result. However, our technique can still check equivalence and inclusion for regular expressions that no existing tool can handle. To answer **Q3, bisimulation and simulation algorithms for SRAs only scale to small numbers of registers.**

6 Conclusions

In this paper we have presented *Symbolic Register Automata*, a novel class of automata that can handle complex alphabet theories while allowing symbol comparisons for equality. SRAs encompass – and are strictly more powerful – than both Register and Symbolic Automata. We have shown that they enjoy the same closure and decidability properties of the former, despite the presence of arbitrary guards on transitions, which are not allowed by RAs. Via a comprehensive set of experiments, we have concluded that SRAs are vastly more succinct than SFAs and membership is efficient on large inputs. Decision procedures do not scale well in the number of registers, which is already the case for basic RAs.

Related Work. RAs were first introduced in [17]. There is an extensive literature on register automata, their formal languages and decidability properties [7, 13, 21, 22, 25], including variants with *global freshness* [20, 27] and totally ordered data [4, 14]. SRAs are based on the original model of [17], but are much more expressive, due to the presence of guards from an arbitrary decidable theory.

In recent work, variants over richer theories have appeared. In [9] RA over rationals were introduced. They allow for a restricted form of linear arithmetic among registers (RAs with arbitrary linear arithmetic subsume two-counter automata, hence are undecidable). SRAs do not allow for operations on registers, but encompass a wider range of theories without any loss in decidability. Moreover, [9] does not study Boolean closure properties. In [8, 16], RAs allowing guards over a range of theories – including (in)equality, total orders and increments/sums – are studied. Their focus is different than ours as they are interested primarily in *active learning* techniques, and several restrictions are placed on models for the purpose of the learning process. We can also relate SRAs with *Quantified Event Automata* [2], which allow for guards and assignments to registers on transitions. However, in QEA guards can be arbitrary, which could lead to several problems, e.g. undecidable equivalence.

Symbolic automata were first introduced in [28] and many variants of them have been proposed [12]. The one that is closer to SRAs is Symbolic Extended Finite Automata (SEFA) [11]. SEFAs are SFAs in which transitions can read more than one character at a time. A transition of arity k reads k symbols which are consumed if they satisfy the predicate $\varphi(x_1, \dots, x_k)$. SEFAs allow arbitrary k -ary predicates over the input theory, which results in most problems being undecidable (e.g., equivalence and intersection emptiness) and in the model not being closed under Boolean operations. Even when deterministic, SEFAs are not closed under union and intersection. In terms of expressiveness, SRAs and SEFAs are incomparable. SRAs can only use equality, but can compare symbols at arbitrary points in the input while SEFAs can only compare symbols within a constant window, but using arbitrary predicates.

Several works study matching techniques for extended regular expressions [3, 5, 18, 24]. These works introduce automata models with ad-hoc features for extended regular constructs – including back-references – but focus on efficient matching, without studying closure and decidability properties. It is also worth noting that SRAs are not limited to alphanumeric or finite alphabets. On the negative side, SRAs cannot express capturing groups of an unbounded length, due to the finitely many registers. This limitation is essential for decidability.

Future Work. In [21] a polynomial algorithm for checking language equivalence of deterministic RAs is presented. This crucially relies on closure properties of symbolic bisimilarity, some of which are lost for SRAs. We plan to investigate whether this algorithm can be adapted to our setting. Extending SRAs with more complex comparison operators other than equality (e.g., a total order $<$) is an interesting research question, but most extensions of the model quickly lead to undecidability. We also plan to study active automata learning for SRAs, building on techniques for SFAs [1], RAs [6, 8, 16] and nominal automata [19].

References

- Argyros, G., D’Antoni, L.: The learnability of symbolic automata. In: CAV, pp. 427–445 (2018)
- Barringer, H., Falcone, Y., Havelund, K., Reger, G., Rydeheard, D.E.: Quantified event automata: towards expressive and efficient runtime monitors. In: FM, pp. 68–84 (2012)
- Becchi, M., Crowley, P.: Extending finite automata to efficiently match perl-compatible regular expressions. In: CoNEXT, pp. 25 (2008)
- Benedikt, M., Ley, C., Puppis, G.: What you must remember when processing data words. In: AMW (2010)
- Bispo, J., Sourdis, I., Cardoso, J.M.P., Vassiliadis, S.: Regular expression matching for reconfigurable packet inspection. In: FPT, pp. 119–126 (2006)
- Bollig, B., Habermehl, P., Leucker, M., Monmege, B.: A fresh approach to learning register automata. In: DLT, pp. 118–130 (2013)
- Cassel, S., Howar, F., Jonsson, B., Merten, M., Steffen, B.: A succinct canonical register automaton model. J. Log. Algebr. Meth. Program. **84**(1), 54–66 (2015)

8. Cassel, S., Howar, F., Jonsson, B., Steffen, B.: Active learning for extended finite state machines. *Formal Asp. Comput.* **28**(2), 233–263 (2016)
9. Chen, Y., Lengál, O., Tan, T., Wu, Z.: Register automata with linear arithmetic. In: LICS, pp. 1–12 (2017)
10. D'Antoni, L., Ferreira, T., Sammartino, M., Silva, A.: Symbolic register automata. CoRR, abs/1811.06968 (2019). <http://arxiv.org/abs/1811.06968>
11. D'Antoni, L., Veànes, M.: Extended symbolic finite automata and transducers. *Formal Meth. Syst. Des.* **47**(1), 93–119 (2015)
12. D'Antoni, L., Veànes, M.: The power of symbolic automata and transducers. In: CAV, pp. 47–67 (2017)
13. Demri, S., Lazic, R.: LTL with the freeze quantifier and register automata. *ACM Trans. Comput. Log.* **10**(3), 16:1–16:30 (2009)
14. Figueira, D., Hofman, P., Lasota, S.: Relating timed and register automata. *Math. Struct. Comput. Sci.* **26**(6), 993–1021 (2016)
15. Grigore, R., Distefano, D., Petersen, R.L., Tzevelekos, N.: Runtime verification based on register automata. In: TACAS, pp. 260–276 (2013)
16. Isberner, M., Howar, F., Steffen, B.: Learning register automata: from languages to program structures. *Mach. Learn.* **96**(1–2), 65–98 (2014)
17. Kaminski, M., Francez, N.: Finite-memory automata. *Theor. Comput. Sci.* **134**(2), 329–363 (1994)
18. Komendantsky, V.: Matching problem for regular expressions with variables. In: Loidl, H.-W., Peña, R. (eds.) TFP 2012. LNCS, vol. 7829, pp. 149–166. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40447-4_10
19. Moerman, J., Sammartino, M., Silva, A., Klin, B., Szynwelski, M.: Learning nominal automata. In: POPL, pp. 613–625 (2017)
20. Murawski, A.S., Ramsay, S.J., Tzevelekos, N.: Bisimilarity in fresh-register automata. In: LICS, pp. 156–167 (2015)
21. Murawski, A.S., Ramsay, S.J., Tzevelekos, N.: Polynomial-time equivalence testing for deterministic fresh-register automata. In: MFCS, pp. 72:1–72:14 (2018)
22. Neven, F., Schwentick, T., Vianu, V.: Finite state machines for strings over infinite alphabets. *ACM Trans. Comput. Log.* **5**(3), 403–435 (2004)
23. RegExLib. Regular expression library (2017). <http://regexlib.com/>
24. Reidenbach, D., Schmid, M.L.: A polynomial time match test for large classes of extended regular expressions. In: Domaratzki, M., Salomaa, K. (eds.) CIAA 2010. LNCS, vol. 6482, pp. 241–250. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18098-9_26
25. Sakamoto, H., Ikeda, D.: Intractability of decision problems for finite-memory automata. *Theor. Comput. Sci.* **231**(2), 297–308 (2000)
26. SVPAlib: Symbolic automata library (2018). <https://github.com/lorisdanto/symbolicautomata>
27. Tzevelekos, N.: Fresh-register automata. In: POPL, pp. 295–306 (2011)
28. Veànes, M., Halleux, P.D., Tillmann, N.: Rex: symbolic regular expression explorer. In: ICST, pp. 498–507 (2010)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Abstraction Refinement Algorithms for Timed Automata

Victor Roussanaly, Ocan Sankur,
and Nicolas Markey^(✉)

Univ Rennes, Inria, CNRS, IRISA, Rennes, France
nmarkey@irisa.fr



Abstract. We present abstraction-refinement algorithms for model checking safety properties of timed automata. The abstraction domain we consider abstracts away zones by restricting the set of clock constraints that can be used to define them, while the refinement procedure computes the set of constraints that must be taken into consideration in the abstraction so as to exclude a given spurious counterexample. We implement this idea in two ways: an enumerative algorithm where a lazy abstraction approach is adopted, meaning that possibly different abstract domains are assigned to each exploration node; and a symbolic algorithm where the abstract transition system is encoded with Boolean formulas.

1 Introduction

Model checking [4, 10, 12, 26] is an automated technique for verifying that the set of behaviors of a computer system satisfies a given property. Model-checking algorithms explore finite-state automata (representing the system under study) in order to decide if the property holds; if not, the algorithm returns an explanation. These algorithms have been extended to verify real-time systems modelled as timed automata [2, 3], an extension of finite automata with clock variables to measure and constrain the amount of time elapsed between occurrences of transitions. The state-space exploration can be done by representing clock constraints efficiently using convex polyhedra called *zones* [8, 9]. Algorithms based on this data structure have been implemented in several tools such as Uppaal [7], and have been applied in various industrial cases.

The well-known issue in the applications of model checking is the *state-space explosion* problem: the size of the state space grows exponentially in the size of the description of the system. There are several sources for this explosion: the system might be made of the composition of several subsystems (such as a distributed system), it might contain several discrete variables (such as in a piece of software), or it might contain a number of real-valued clocks as in our case.

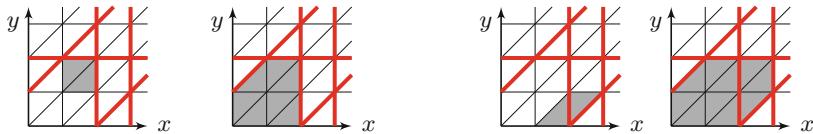
This work was funded by ANR project Ticktac (ANR-18-CE40-0015) and by ERC grant EQUALIS (StG-308087).

Numerous attempts have been made to circumvent this problem. Abstraction is a generic approach that consists in simplifying the model under study, so as to make it easier to verify [13]. *Existential* abstraction may only add extra behaviors, so that when a safety property holds in an abstracted model, it also holds in the original model; if on the other hand a safety property fails to hold, the model-checking algorithms return a witness trace exhibiting the non-safe behaviour: this either invalidates the property on the original model, if the trace exists in that model, or gives information about how to automatically refine the abstraction. This approach, named CEGAR (counter-example guided abstraction refinement) [11], was further developed and used, for instance, in software verification (BLAST [20], SLAM [5], ...).

The CEGAR approach has been adapted to timed automata, e.g. in [14, 18], but the abstractions considered there only consist in removing clocks and discrete variables, and adding them back during refinement. So for most well-designed models, one ends up adding all clocks and variables which renders the method useless. Two notable exceptions are [22], in which the zone extrapolation operators are dynamically adapted during the exploration, and [29], in which zones are refined when needed using interpolants. Both approaches define “exact” abstractions in the sense that they make sure that all traces discovered in the abstract model are feasible in the concrete model at any time.

In this work, we consider a more general setting and study *predicate abstractions* on clock variables. Just like in software model checking, we define abstract state spaces using these predicates, where the values of the clocks and their relations are approximately represented by these predicates. New predicates are generated if needed during the refinement step. We instantiate our approach by two algorithms. The first one is a zone-based enumerative algorithm inspired by the *lazy abstraction* in software model checking [19], where we assign a possibly different abstract domain to each node in the exploration. The second algorithm is based on binary decision diagrams (BDD): by exploiting the observation that a small number of predicates was often sufficient to prove safety properties, we use an efficient BDD encoding of zones similar to one introduced in early work [28].

Let us explain the abstract domains we consider. Assume there are two clock variables x and y . The abstraction we consider consists in restricting the clock



(a) Abstraction of zone $1 \leq x, y \leq 2$ (b) Abstraction of zone $y \leq 1 \wedge 1 \leq x - y \leq 2$

Fig. 1. The abstract domain is defined by the clock constraints shown in thick red lines. In each example, the abstraction of the zone shown on the left (shaded area) is the larger zone on the right. (Color figure online)

constraints that can be used when defining zones. Assume that we only allow to compare x with 2 or 3; that y can only be compared with 2, and $x - y$ can only be compared with -1 or 2. Then any conjunction of constraints one might obtain in this manner will be delimited by the thick red lines in Fig. 1; one cannot define a finer region under this restriction. The figure shows the abstraction process: given a “concrete” zone, its abstraction is the smallest zone which is a superset and is definable under our restriction. For instance, the abstraction of $1 \leq x, y \leq 2$ is $0 \leq x, y \leq 2 \wedge -1 \leq x - y$ (cf. Fig. 1a).

Related Works. We give more detail on zone abstractions in timed automata. Most efforts in the literature have been concentrated in designing zone abstraction operators that are exact in the sense that they preserve the reachability relation between the locations of a timed automaton; see [6]. The idea is to determine bounds on the constants to which a given clock can be compared to in a given part of the automaton, since the clock values do not matter outside these bounds. In [21, 22], the authors give an algorithm where these bounds are dynamically adapted during the exploration, which allows one to obtain coarser abstractions. In [29], the exploration tree contains pairs of zones: a concrete zone as in the usual algorithm, and a coarser abstract zone. The algorithm explores all branches using the coarser zone and immediately refines the abstract zone whenever an edge which is disabled in the concrete zone is enabled. In [17], a CEGAR loop was used to solve timed games by analyzing strategies computed for each abstract game. The abstraction consisted in collapsing locations.

Some works have adapted the abstraction-refinement paradigm to timed automata. In [14], the authors apply “localization reduction” to timed automata within an abstraction-refinement loop: they abstract away clocks and discrete variables, and only introduce them as they are needed to rule out spurious counterexamples. A more general but similar approach was developed in [18]. In [31], the authors adapt the trace abstraction refinement idea to timed automata where a finite automaton is maintained to rule out infeasible edge sequences.

The CEGAR approach was also used recently in the LinAIG framework for verifying linear hybrid automata [1]. In this work, the backward reachability algorithm exploits *don’t-cares* to reduce the size of the Boolean circuits representing the state space. The abstractions consist in enlarging the size of *don’t-cares* to reduce the number of linear predicates used in the representation.

2 Timed Automata and Zones

2.1 Timed Automata

Given a finite set of clocks \mathcal{C} , we call *valuations* the elements of $\mathbb{R}_{\geq 0}^{\mathcal{C}}$. For a clock valuation v , a subset $R \subseteq \mathcal{C}$, and a non-negative real d , we denote with $v[R \leftarrow d]$ the valuation w such that $w(x) = v(x)$ for $x \in \mathcal{C} \setminus R$ and $w(x) = d$ for $x \in R$, and with $v + d$ the valuation w' such that $w'(x) = v(x) + d$ for all $x \in \mathcal{C}$. We extend these operations to sets of valuations in the obvious way. We write $\mathbf{0}$ for the valuation that assigns 0 to every clock. An *atomic guard* is a formula of

the form $x \prec k$ or $x - y \prec k$ with $x, y \in \mathcal{C}$, $k \in \mathbb{N}$, and $\prec \in \{<, \leq, >, \geq\}$. A *guard* is a conjunction of atomic guards. A valuation v satisfies a guard g , denoted $v \models g$, if all atomic guards hold true when each $x \in \mathcal{C}$ is replaced with $v(x)$. Let $\llbracket g \rrbracket = \{v \in \mathbb{R}_{\geq 0}^{\mathcal{C}} \mid v \models g\}$ denote the set of valuations satisfying g . We write $\Phi_{\mathcal{C}}$ for the set of guards built on \mathcal{C} .

A *timed automaton* \mathcal{A} is a tuple $(\mathcal{L}, \text{Inv}, \ell_0, \mathcal{C}, E)$, where \mathcal{L} is a finite set of locations, $\text{Inv}: \mathcal{L} \rightarrow \Phi_{\mathcal{C}}$ defines location invariants, \mathcal{C} is a finite set of clocks, $E \subseteq \mathcal{L} \times \Phi_{\mathcal{C}} \times 2^{\mathcal{C}} \times \mathcal{L}$ is a set of edges, and $\ell_0 \in \mathcal{L}$ is the initial location. An edge $e = (\ell, g, R, \ell')$ is also written as $\ell \xrightarrow{g, R} \ell'$. For any location ℓ , we let $E(\ell)$ denote the set of edges leaving ℓ .

A *configuration* of \mathcal{A} is a pair $q = (\ell, v) \in \mathcal{L} \times \mathbb{R}_{\geq 0}^{\mathcal{C}}$ such that $v \models \text{Inv}(\ell)$. A *run* of \mathcal{A} is a sequence $q_1 e_1 q_2 e_2 \dots q_n$ where for all $i \geq 1$, $q_i = (\ell_i, v_i)$ is a configuration, and either $e_i \in \mathbb{R}_{>0}$, in which case $q_{i+1} = (\ell_i, v_i + e_i)$, or $e_i = (\ell_i, g_i, R_i, \ell_{i+1}) \in E$, in which case $v_i \models g_i$ and $q_{i+1} = (\ell_{i+1}, v_i[R_i \leftarrow 0])$. A *path* is a sequence of edges with matching endpoint locations.

2.2 Zones and DBMs

Several tools for timed automata implement algorithms based on *zones*, which are particular polyhedra definable with clock constraints. Formally, a zone Z is a subset of $\mathbb{R}_{\geq 0}^{\mathcal{C}}$ definable by a guard in $\Phi_{\mathcal{C}}$.

We recall a few basic operations defined on zones. First, the intersection $Z \cap Z'$ of two zones Z and Z' is clearly a zone. Given a zone Z , the set of time-successors of Z , defined as $Z\uparrow = \{v + t \in \mathbb{R}_{\geq 0}^{\mathcal{C}} \mid t \in \mathbb{R}_{\geq 0}, v \in Z\}$, is easily seen to be a zone; similarly for time-predecessors $Z\downarrow = \{v \in \mathbb{R}_{\geq 0}^{\mathcal{C}} \mid \exists t \geq 0. v + t \in Z\}$. Given $R \subseteq \mathcal{C}$, we let $\text{Reset}_R(Z)$ be the zone $\{v[R \leftarrow 0] \in \mathbb{R}_{\geq 0}^{\mathcal{C}} \mid v \in Z\}$, and $\text{Free}_x(Z) = \{v' \in \mathbb{R}_{\geq 0}^{\mathcal{C}} \mid \exists v \in Z, d \in \mathbb{R}_{\geq 0}, v' = v[x \leftarrow d]\}$.

Zones can be represented as *difference-bound matrices* (DBM) [8, 15]. Let $\mathcal{C}_0 = \mathcal{C} \cup \{0\}$, where 0 is an extra symbol representing a special clock variable whose value is always 0. A DBM is a $|\mathcal{C}_0| \times |\mathcal{C}_0|$ -matrix taking values in $(\mathbb{Z} \times \{<, \leq\}) \cup \{(+\infty, <)\}$. Intuitively, cell (x, y) of a DBM M stores a pair (d, \prec) representing an upper bound on the difference $x - y$. For any DBM M , we let $\llbracket M \rrbracket$ denote the zone it defines.

While several DBMs can represent the same zone, each zone admits a *canonical* representation, which is obtained by storing the tightest clock constraints defining the zone. This canonical representation can be obtained by computing shortest paths in a graph where the vertices are clocks and the edges weighted by clock constraints, with natural addition and comparison of elements of $(\mathbb{Z} \times \{<, \leq\}) \cup \{(+\infty, <)\}$. This graph has a negative cycle if, and only if, the associated DBM represents the empty zone.

All the operations on zones can be performed efficiently (in $O(|\mathcal{C}_0|^3)$) on their associated DBMs while maintaining reduced form. For instance, the intersection $N = Z \cap Z'$ of two canonical DBMs Z and Z' can be obtained by first computing the DBM $M = \min(Z, Z')$ such that $M(x, y) = \min\{Z(x, y), Z'(x, y)\}$ for all $(x, y) \in \mathcal{C}_0^2$, and then turning M into canonical form. We refer to [8] for

full details. By a slight abuse of notation, we use the same notations for DBMs as for zones, writing e.g. $M' = M\uparrow$, where M and M' are reduced DBMs such that $\llbracket M' \rrbracket = \llbracket M \rrbracket \uparrow$. Given an edge $e = (\ell, g, R, \ell')$, and a zone Z , we define $\text{Post}_e(Z) = \text{Inv}(\ell') \cap (g \cap \text{Reset}_R(Z))\uparrow$, and $\text{Pre}_e(Z) = (g \cap \text{Free}_R(\text{Inv}(\ell') \cap Z))\downarrow$. For a path $\rho = e_1 e_2 \dots e_n$, we define Post_ρ and Pre_ρ by iteratively applying Post_{e_i} and Pre_{e_i} respectively.

2.3 Clock-Predicate Abstraction and Interpolation

For all clocks x and y in \mathcal{C}_0 , we consider a finite set $\mathcal{D}_{x,y} \subseteq \mathbb{N} \times \{\leq, <\}$, and gather these in a table $\mathcal{D} = (\mathcal{D}_{x,y})_{x,y \in \mathcal{C}_0}$. \mathcal{D} is the *abstract domain* which restricts zones to be defined only using constraints of the form $x - y \prec k$ with $(k, \prec) \in \mathcal{D}_{x,y}$, as seen earlier. Let us call \mathcal{D} the *concrete domain* if $\mathcal{D}_{x,y} = \mathbb{N} \times \{\leq, <\}$ for all $x, y \in \mathcal{C}_0$. A zone Z is \mathcal{D} -definable if there exists a DBM D such that $Z = \llbracket D \rrbracket$ and $D(x, y) \in \mathcal{D}_{x,y}$ for all $x, y \in \mathcal{C}_0$. Note that we do not require this witness DBM D to be reduced; the reduction of such a DBM might introduce additional values. We say that domain \mathcal{D}' is a *refinement* of \mathcal{D} if for all $x, y \in \mathcal{C}_0$, we have $\mathcal{D}_{x,y} \subseteq \mathcal{D}'_{x,y}$.

An abstract domain \mathcal{D} induces an *abstraction function* $\alpha_{\mathcal{D}} : 2^{\mathbb{R}_{\geq 0}^{\mathcal{C}}} \rightarrow 2^{\mathbb{R}_{\geq 0}^{\mathcal{C}}}$ where $\alpha_{\mathcal{D}}(Z)$ is the smallest \mathcal{D} -definable zone containing Z . For any reduced DBM D , $\alpha_{\mathcal{D}}(\llbracket D \rrbracket)$ can be computed by setting $D'(x, y) = \min\{(k, \prec) \in \mathcal{D}_{x,y} \mid D(x, y) \leq (k, \prec)\}$ (with $\min \emptyset = (\infty, <)$).

An *interpolant* for a pair of zones (Z_1, Z_2) with $Z_1 \cap Z_2 = \emptyset$ is a zone Z_3 with $Z_1 \subseteq Z_3$ and $Z_3 \cap Z_2 = \emptyset$ ¹ [29]. We use interpolants to refine our abstractions; in order not to add too many new constraints when refining, our aim is to find *minimal interpolants*: define the density of a DBM D as $d(D) = \#\{(x, y) \in \mathcal{C}_0^2 \mid D(x, y) \neq (\infty, <)\}$. Notice that while any pair of disjoint convex polyhedra can be separated by hyperplanes, not all pairs of disjoint zones admit interpolants of density 1; this is because not all (half-spaces delimited by) hyperplanes are zones. Still, we can bound the density of a minimal interpolant:

Lemma 1. *For any pair of disjoint, non-empty zones (A, B) , there exists an interpolant of density less than or equal to $|\mathcal{C}_0|/2$.*

By adapting the algorithm of [29] for computing interpolants, we can compute minimal interpolants efficiently:

Proposition 2. *Computing a minimal interpolant can be performed in $O(|\mathcal{C}|^4)$.*

3 Enumerative Algorithm

The first type of algorithm we present is a zone-based enumerative algorithm based on the clock-predicate abstractions. Let us first describe the overall

¹ It is sometimes also required that the interpolant only involves clocks that have non-trivial constraints in both Z_1 and Z_2 . We do not impose this requirement in our definition, but it will hold true in the interpolants computed by our algorithm.

algorithm in Algorithm 1, which is a typical abstraction-refinement loop. We then explain how the abstract reachability and refinement procedures are instantiated.

Algorithm 1. Enumerative algorithm checking the reachability of a target location ℓ_T .

```

Input:  $\mathcal{A} = (\mathcal{L}, \text{Inv}, \ell_0, \mathcal{C}, E)$ ,  $\ell_T$ 
1 Initialize  $\mathcal{D}_0$ ;
2 wait := {node( $\ell_0$ ,  $\mathbf{0}\uparrow$ ,  $\mathcal{D}_0$ )};
3 passed :=  $\emptyset$ ;
4 while do
5    $\pi := \text{AbsReach}(\mathcal{A}, \text{wait},$ 
     $\text{passed}, \ell_T)$ ;
6   if  $\pi = \emptyset$  then
7     return Not reachable;
8   else
9     if trace  $\pi$  is feasible then
10      return Reachable;
11    else
12      Refine( $\pi$ , wait, passed);
13
14
15 return Not reachable;
```

Algorithm 2. AbsReach

```

Input:  $(\mathcal{L}, \text{Inv}, \ell_0, \mathcal{C}, E)$ , wait, passed,
 $\ell_T$ 
1 while wait  $\neq \emptyset$  do
2    $n := \text{wait.pop}()$ ;
3   if  $n.\ell = \ell_T$  then
4     return Trace from root to  $n$ ;
5   if  $\exists n' \in \text{passed}$  such that  $n.\ell =$ 
     $n'.\ell \wedge n.Z \subseteq n'.Z$  then
6      $n.\text{covered} := n'$ ;
7   else
8      $n.Z := \alpha(n.Z, n)$ ;
9     passed.add( $n$ );
10    for  $e = (\ell, g, R, \ell') \in E(n.\ell)$ 
    s.t.  $Z' := \text{Post}_e(n.Z) \neq \emptyset$ 
11    do
12       $\mathcal{D}' := \text{choose-dom}(n, e)$ ;
13       $n' := \text{node}(\ell', Z', \mathcal{D}')$ ;
14       $n'.\text{parent} := n$ ;
        wait.add( $n'$ );
15
16 return  $\emptyset$ ;
```

The initialization at line 1 chooses an abstract domain for the initial state, which can be either empty (thus the coarsest abstraction) or defined according to some heuristics. The algorithm maintains the `wait` and `passed` lists that are used in the forward exploration. As usual, the `wait` list can be implemented as a stack, a queue, or another priority list that determines the search order. The algorithm also uses covering nodes. Indeed if there are two node n and n' , with $n \in \text{passed}$, $n' \in \text{wait}$, $n.\ell = n'.\ell$, and $n'.z \subseteq n.Z$, then we know that every location reachable from n' is also reachable from n . Since we have already explored n and we generated its successors, there is no need to explore the successors of n' . The algorithm explicitly creates an exploration tree: line 2 creates a node containing location ℓ_0 , zone $\mathbf{0}\uparrow$, and the abstract domain \mathcal{D}_0 as the root of our tree, and adds this to the `wait` list. More details on the tree are given in the next subsection. Procedure `AbsReach` then looks for a trace to the target location ℓ_T . If such a trace exists, line 9 checks its feasibility. Here π is a sequence of node and edges of \mathcal{A} . The feasibility check is done by computing predecessors with zones starting from the final state, without using the abstraction function. If the last zone intersects our initial zone, this means that the trace is feasible. More details are given in Sect. 3.2.

3.1 Abstract Forward Reachability: **AbsReach**

We give a generic algorithm independently from the implementations of the abstraction functions and the refinement procedure.

Algorithm 2 describes the reachability procedure under a given abstract domain \mathcal{D} . It is similar to the standard forward reachability algorithm using a **wait-list** and a **passed-list**. We explicitly create an exploration tree where the leaves are nodes in **wait**, covered nodes, or nodes that have no non-empty successors. Each node n contains the fields ℓ, Z which are labels describing the current location and zone; field **covered** points to a node covering the current node (it is undefined if the current node is not (known to be) covered); field **parent** points to the parent node in the tree (it is undefined for the root); and field \mathcal{D} is the abstract domain associated with the node. Thus, the algorithm uses a possibly different abstract domain for each node in the exploration tree.

The difference of our algorithm w.r.t. the standard reachability can be seen at lines 8 and 11. At line 8, we apply the abstraction function to the zone taken from the **wait-list** before adding it to the **passed-list**. The abstraction function α is a function of a zone Z and a node n . This allows one to define variants with different dependencies; for instance, α might depend on the abstract domain $n.\mathcal{D}$ at the current node, but it can also use other information available in n or on the path ending in n . For now, it is best to think of α simply as $Z \mapsto \alpha_{n,\mathcal{D}}(Z)$. At line 11, the function **choose-dom** chooses an abstract domain for the node n' . The domain could be chosen global for all nodes, or local to each node. A good trade-off, which we used in our experiments, is to have domains associated with locations of the timed automaton.

Remark 1. Note that we use the abstraction function when the node is inserted in the **passed** list. This is because we want the node to contain the smallest zone possible when we test whether the node is covered. We only need to use the abstracted zone when we compute its successor and when we test whether the node is covering. This allows us to store a unique zone.

As a first step towards proving correctness of our algorithm, we show that the following property is preserved by Algorithm **AbsReach**:

For all nodes n in **passed**, for all edges e from $n.\ell$, if $\text{Post}_e(n.Z) \neq \emptyset$, then n has a child n' such that $\text{Post}_e(n.Z) \subseteq n'.Z$. If n' is in **passed**, then we also have $\alpha_{n',\mathcal{D}}(\text{Post}_e(n.Z)) \subseteq n'.Z$. (1)

Lemma 3. *Algorithm **AbsReach** preserves Property (1).*

Note that although we use inclusion in Property (1), **AbsReach** would actually preserve equality of zones, but we will not always have equality before running **AbsReach**. This is because **Refine** might change the zones of some nodes without updating the zones of all their descendants.

3.2 Refinement: Refine

We now describe our refinement procedure **Refine**. Let us now assume that AbsReach returns $\pi = A_1 \xrightarrow{\sigma_1} A_2 \xrightarrow{\sigma_2} \dots \xrightarrow{\sigma_{k-1}} A_k$, and write \mathcal{D}_i for the domain associated with each A_i . We write C_1 for the initial concrete zone, and for $i < k$, we define $C_{i+1} = \text{Post}_{\sigma_i}(A_i)$. We also note $Z_k = A_k$ and for $i < k$, $Z_i = \text{Pre}_{\sigma_i}(Z_{i+1}) \cap A_i$. Then π is not feasible if, and only if, $\text{Post}_{\sigma_1 \dots \sigma_k}(C_1) = \emptyset$, or equivalently $\text{Pre}_{\sigma_1 \dots \sigma_k}(A_k) \cap C_1 = \emptyset$. Since for all $i < k$, it holds $C_i \subseteq A_{i+1}$, we have that π is not feasible if, and only if, $\exists i \leq k. C_i \cap Z_i = \emptyset$. We illustrate this on Fig. 2.

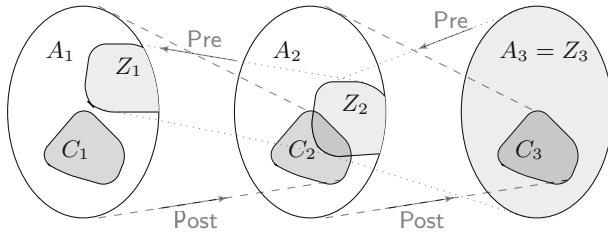


Fig. 2. Spurious counter-example: $Z_1 \cap C_1 = \emptyset$

Let us assume that π is not feasible. Let us denote by i_0 the maximal index such that $C_{i_0} \cap Z_{i_0} = \emptyset$. This index also has the property that for all $j < i_0$, we have $Z_j = \emptyset$ and $Z_{i_0} \neq \emptyset$. Once we have identified this trace as spurious by computing the Z_j , we have two possibilities:

- if $Z_{i_0} \cap \alpha_{\mathcal{D}_{i_0}}(C_{i_0}) \neq \emptyset$: this means that we can reach A_k from $\alpha_{\mathcal{D}_{i_0}}(C_{i_0})$ but not from C_{i_0} . In other words, our abstraction is too coarse and we must add some values to \mathcal{D}_{i_0} so that $Z_{i_0} \cap \alpha_{\mathcal{D}_{i_0}}(C_{i_0}) = \emptyset$. Those values are found by computing the interpolant of Z_{i_0} and C_{i_0}
- Otherwise it means that $\alpha_{\mathcal{D}_{i_0}}(C_{i_0})$ cannot reach A_k and the only reason the trace exists is because either \mathcal{D}_{i_0} or A_{i_0-1} has been modified at some point and A_{i_0} was not modified accordingly.

We can then update the values of C_i for $i > i_0$ and repeat the process until we reach an index j_0 such that $C_{j_0} = \emptyset$. We then have modified the nodes n_{i_0}, \dots, n_{j_0} and knowing that $n_{j_0}.Z = \emptyset$, we can delete it and all of its descendants. Since some of the descendants of n_{i_0} have not been modified, this might cause some refinements of the first type in the future. In order to ensure termination, we sometimes have to cut a subtree from a node in $n_{i_0}, \dots, n_{j_0-1}$ and reinsert it in the `wait` list to restart the exploration from there. We call this action `cut`, and we can use several heuristics to decide when to use it. In the rest of this paper we will use the following heuristics: we perform `cut` on the first node of $n_{i_0} \dots n_{j_0}$ that is covered by some other node. Since this node is covered, we know that we will not restart the exploration from this node, or that the

node was covered by one of its descendant. If none of these nodes are covered, we delete n_{j_0} and its descendants. Other heuristics are possible, for instance applying cut on n_{i_0} . We found that the above heuristics was the most efficient in our experiments.

Lemma 4. *Pick a node n , and let $Y = n.Z$. Then after running Refine, either node n is deleted, or it holds $n.Z \subseteq Y$. In other words, the zone of a node can only be reduced by Refine.*

It follows that Refine also preserves Property (1), so that:

Lemma 5. *Algorithm 1 satisfies Property (1).*

We can then prove that our algorithm correctly decides the reachability problem and always terminates.

Theorem 6. *Algorithm 1 terminates and is correct.*

4 Symbolic Algorithm

4.1 Boolean Encoding of Zones

We now present a symbolic algorithm that represents abstract states using Boolean formulas. Let $\mathbb{B} = \{0, 1\}$, and \mathcal{V} be a set of variables. A Boolean formula f that uses variables from set $X \subseteq \mathcal{V}$ will be written $f(X)$ to make the dependency explicit; we sometimes write $f(X, Y)$ in place of $f(X \cup Y)$. Such a formula represents a set $\llbracket f \rrbracket = \{v \in \mathbb{B}^{\mathcal{V}} \mid v \models f\}$. We consider primed versions of all variables; this will allow us to write formulas relating two valuations. For any subset $X \subseteq \mathcal{V}$, we define $X' = \{p' \mid p \in X\}$.

A *literal* is either p or $\neg p$ for a variable p . Given a set X of variables, an X -*minterm* is the conjunction of literals where each variable of X appears exactly once. X -minterms can be seen as elements of \mathbb{B}^X . Given a vector of Boolean formulas $Y = (Y_x)_{x \in X}$, formula $f[Y/X]$ is the *substitution of X by Y in f* , obtained by replacing each $x \in X$ with the formula Y_x . The positive cofactor of $f(X)$ by x is $\exists x. (x \wedge f(X))$, and its negative cofactor is $\exists x. (\neg x \wedge f(X))$.

Let us define a generic operator *post* that computes successors of a set $S(X, Y)$ given a relation $R(X, X')$ (here, Y designates any set of variables on which S might depend outside of X): $\text{post}_R(S(X, Y)) = (\exists X.S(X, Y) \wedge R(X, X'))[X/X']$. Similarly, we set $\text{pre}_R(S(X, Y)) = (\exists X'.S(X, Y)[X'/X] \wedge R(X, X'))$, which computes the predecessors of $S(X, Y)$ by the relation R [24].

Clock Predicate Abstraction. We fix a total order \triangleleft on \mathcal{C}_0 . In this section, abstract domains are defined as $\mathcal{D} = (\mathcal{D}_{x,y})_{x \triangleleft y \in \mathcal{C}_0}$, that is only for pairs $x \triangleleft y$. In fact, constraints of the form $x - y \leq k$ with $x \triangleright y$ are encoded using the negation of $y - x < -k$ since $(x - y \leq k) \Leftrightarrow \neg(y - x < -k)$. We thus define $\mathcal{D}_{x,y} = -\mathcal{D}_{y,x}$ for all $x \triangleright y$.

For $x, y \in \mathcal{C}_0$, let $\mathcal{P}_{x,y}$ denote the set of *clock predicates associated to $\mathcal{D}_{x,y}$* :

$$\mathcal{P}_{x,y}^{\mathcal{D}} = \{P_{x-y \prec k} \mid (k, \prec) \in \mathcal{D}_{x,y}\}.$$

Let $\mathcal{P}^{\mathcal{D}} = \cup_{x,y \in \mathcal{C}_0} \mathcal{P}_{x,y}$ denote the set of all clock predicates associated with \mathcal{D} (we may omit the superscript \mathcal{D} when it is clear). For all $(x, y) \in \mathcal{C}_0^2$ and $(k, \prec) \in \mathcal{D}_{x,y}$, we denote by $p_{x-y \prec k}$ the literal $P_{x-y \prec k}$ if $x \triangleleft y$, and $\neg P_{y-x \prec^{-1}-k}$ otherwise (where $\leq^{-1} = <$ and $<^{-1} = \leq$). We also consider a set \mathcal{B} of Boolean variables used to encode locations. Overall, the state space is described using Boolean formulas on these two types of variables, so states are elements of $\mathbb{B}^{\mathcal{P} \cup \mathcal{B}}$.

Our Boolean encoding of clock constraints and semantic operations follow those of [28] for a concrete domain. We define these however for abstract domains, and show how successor computation and refinement operations can be performed.

Let us define the *clock semantics* of predicate $P_{x-y \leq k}$ as $\llbracket P_{x-y \leq k} \rrbracket_{\mathcal{C}_0} = \{\nu \in \mathbb{R}_{\geq 0}^{\mathcal{C}_0} \mid \nu(x) - \nu(y) \leq k\}$. Since the set \mathcal{C} of clocks is fixed, we may omit the subscript and just write $\llbracket P_{x-y \leq k} \rrbracket$. We define the conjunction, disjunction, and negation as intersection, union, and complement, respectively. Given a \mathcal{P} -minterm $v \in \mathbb{B}^{\mathcal{P}}$, we define $\llbracket v \rrbracket_{\mathcal{D}} = \bigcap_{p \text{ s.t. } v(p)} \llbracket p \rrbracket_{\mathcal{D}} \cap \bigcap_{p \text{ s.t. } \neg v(p)} \llbracket p \rrbracket_{\mathcal{D}}^c$. Thus, negation of a predicate encodes its complement. For a Boolean formula $F(\mathcal{P})$, we set $\llbracket F \rrbracket = \bigcup_{v \in \text{Minterms}(F)} \llbracket v \rrbracket_{\mathcal{D}}$. Intuitively, the minterms of \mathcal{P} define smallest zones of $\mathbb{R}_{\geq 0}^{\mathcal{C}}$ definable using \mathcal{P} . A minterm $v \in \mathbb{B}^{\mathcal{B} \cup \mathcal{P}}$ defines a pair $\llbracket v \rrbracket_{\mathcal{D}} = (l, Z)$ where l is encoded by $v|_{\mathcal{B}}$ and $Z = \llbracket v|_{\mathcal{P}} \rrbracket_{\mathcal{D}}$. A Boolean formula F on $\mathcal{B} \cup \mathcal{P}$ defines a set $\llbracket F \rrbracket_{\mathcal{D}} = \bigcup_{v \in \text{Minterms}(F)} \llbracket v \rrbracket_{\mathcal{D}}$ of such pairs. A minterm v is *satisfiable* if $\llbracket v \rrbracket_{\mathcal{D}} \neq \emptyset$.

An abstract domain \mathcal{D} induces an *abstraction function* $\alpha_{\mathcal{D}}: 2^{\mathbb{R}_{\geq 0}^{\mathcal{C}}} \rightarrow 2^{\mathbb{B}^{\mathcal{P}}}$ with $\alpha_{\mathcal{D}}(Z) = \{v \mid v \in \mathbb{B}^{\mathcal{P}} \text{ and } \llbracket v \rrbracket_{\mathcal{D}} \cap Z \neq \emptyset\}$, from the set of zones to the set of subsets of Boolean valuations on \mathcal{P} . We define the *concretization function* as $\llbracket \cdot \rrbracket_{\mathcal{D}}: 2^{\mathbb{B}^{\mathcal{P}}} \rightarrow 2^{\mathbb{R}_{\geq 0}^{\mathcal{C}}}$. The pair $(\alpha_{\mathcal{D}}, \llbracket \cdot \rrbracket_{\mathcal{D}})$ is a Galois connection, and $\llbracket \alpha_{\mathcal{D}}(Z) \rrbracket_{\mathcal{D}}$ is the most precise abstraction of Z in the domain induced by \mathcal{D} . Notice that $\alpha_{\mathcal{D}}$ is non-convex in general: for instance, if the clock predicates are $x \leq 2, y \leq 2$, then the set defined by the constraint $x = y$ maps to $(p_{x \leq 2} \wedge p_{y \leq 2}) \vee (\neg p_{x \leq 2} \wedge \neg p_{y \leq 2})$.

4.2 Reduction and Successor Computation

We now define the reduction operation, which is similar to the reduction of DBMs. The idea is to eliminate unsatisfiable minterms from a given Boolean formula. For example, we would like to make sure that in all minterms, if $p_{x-y \leq 1}$ holds, then so does $p_{x-y \leq 2}$, when both are available predicates. Another issue is to eliminate minterms that are unsatisfiable due to triangle inequality. This is similar to the shortest path computation used to turn DBMs in canonical form.

Example 1. Given predicates $\mathcal{P} = \{p_{x-y \leq 1}, p_{y-z \leq 1}, p_{x-z \leq 2}\}$, the formula $p_{x-y \leq 1} \wedge p_{y-z \leq 1}$ is not reduced since it contains the unsatisfiable minterm

$p_{x-y \leq 1} \wedge p_{y-z \leq 1} \wedge \neg p_{x-z \leq 2}$. However, the same formula is reduced if $\mathcal{P} = \{p_{x-y \leq 1}, p_{y-z \leq 1}\}$.

In this paper, we use limited reduction, since reductions are the most expensive operations in our algorithms. The following formula corresponds to 2-reduction, which intuitively amounts to applying shortest paths for paths of lengths 1 and 2:

$$\bigwedge_{\substack{(x,y) \in \mathcal{C}_0^2 \\ (k,\prec) \in \mathcal{D}_{x,y}}} \left[p_{x-y \prec k} \leftarrow \left(\bigvee_{\substack{(l_1, \prec_1) \in \mathcal{D}_{x,y} \\ (l_1, \prec_1) \leq (k, \prec)}} p_{x-y \prec_1 l_1} \vee \bigvee_{\substack{z \in \mathcal{C}_0, (l_1, \prec_1) \in \mathcal{D}_{x,z}, \\ (l_2, \prec_2) \in \mathcal{D}_{z,y} \\ (l_1, \prec_1) + (l_2, \prec_2) \leq (k, \prec)}} p_{x-z \prec l} \wedge p_{z-y \prec' l'} \right) \right]$$

Lemma 7. *For all formulas $S(\mathcal{P})$, we have $\llbracket S \rrbracket_{\mathcal{D}} = \llbracket \text{reduce}_{\mathcal{D}}^2(S) \rrbracket_{\mathcal{D}}$ and all minterms of $\text{reduce}_{\mathcal{D}}^2(S)$ are 2-reduced.*

Since 2-reduction does not consider shortest paths of all lengths, there are, in general, 2-reduced unsatisfiable minterms. Nevertheless, any abstraction can be refined so that the updated 2-reduction eliminates a given unsatisfiable minterm:

Lemma 8. *Let $v \in \mathbb{B}^{\mathcal{P}^D}$ be a minterm such that $v \models \text{reduce}_{\mathcal{D}}^2$ and $\llbracket v \rrbracket = \emptyset$. One can compute in polynomial time a refinement $\mathcal{D}' \supset \mathcal{D}$ such that $v \not\models \text{reduce}_{\mathcal{D}'}^2$.*

We now explain how successor computation is realized in our encoding. For a guard g , assume we have computed an abstraction $\alpha_{\mathcal{D}}(g)$ in the present abstract domain. For each transition $\sigma = (\ell_1, g, R, \ell_2)$, let us define the formula $T_{\sigma} = \ell_1 \wedge \alpha_{\mathcal{D}}(g)$. We show how each basic operation on zones can be computed in our BDD encoding. In our algorithm, all formulas $A(\mathcal{B}, \mathcal{P})$ representing sets of states are assumed to be reduced, that is, $A(\mathcal{B}, \mathcal{P}) \subseteq \text{reduce}_{\mathcal{D}}^2(A(\mathcal{B}, \mathcal{P}))$.

The intersection operation is simply logical conjunction:

Lemma 9. *For all reduced formulas $A(\mathcal{P})$ and $B(\mathcal{P})$, we have $A(\mathcal{P}) \wedge B(\mathcal{P}) = \alpha_{\mathcal{D}}(\llbracket A(\mathcal{P}) \rrbracket_{\mathcal{D}} \cap \llbracket B(\mathcal{P}) \rrbracket_{\mathcal{D}})$.*

For the time successors, we define $\text{Up}(A(\mathcal{B}, \mathcal{P})) = \text{reduce}(\text{post}_{S_{\text{Up}}}(A(\mathcal{B}, \mathcal{P})))$ where

$$S_{\text{Up}} = \bigwedge_{\substack{x \in \mathcal{C} \\ (k, \prec) \in \mathcal{D}_{x,0}}} (\neg p_{x-0 \prec k} \rightarrow \neg p'_{x-0 \prec k}) \bigwedge_{\substack{x, y \in \mathcal{C}_0, x \neq 0 \\ (k, \prec) \in \mathcal{D}_{x,y}}} (p'_{x-y \prec k} \leftrightarrow p_{x-y \prec k}).$$

Lemma 10. *For any Boolean formula $A(\mathcal{B}, \mathcal{P})$, $\alpha_{\mathcal{D}}(\llbracket A \rrbracket \uparrow) \subseteq \text{Up}(A)$. Moreover, if \mathcal{D} is the concrete domain and A is reduced, then this holds with equality.*

Following similar ideas, we handle clock resets by defining $\text{Reset}_z(A) = \text{reduce}(\text{post}_{S_{\text{Reset}_z}}(A))$, for a (complex) relation S_{Reset_z} to encode how predicates evolve (see the long version [27] of this article for more detailed explanations).

We get:

Lemma 11. *For any Boolean formula $A(\mathcal{B}, \mathcal{P})$, and any clock $z \in \mathcal{C}$, we have $\alpha_{\mathcal{D}}(\text{Reset}_z(\llbracket A \rrbracket_{\mathcal{D}})) \subseteq \text{Reset}_z(A)$. Moreover, if \mathcal{D} is the concrete domain, and A is reduced, then the above holds with equality.*

Algorithm 3. Algorithm SymReach that checks the reachability of a target location l_T in a given abstract domain \mathcal{D} .

```

Input:  $\mathcal{A} = (\mathcal{L}, \text{Inv}, \ell_0, \mathcal{C}, E)$ ,  $\ell_T$ ,  $\mathcal{D}$ 
1 ;
2 next :=  $\text{enc}(l_0) \wedge \alpha_{\mathcal{D}}(\wedge_{x \in \mathcal{C}} x = 0)$ ;
3 layers := [];
4 reachable := false;
5 while ( $\neg$ reachable  $\wedge$  next)  $\neq$  false do
6   | reachable := reachable  $\vee$  next;
7   | next := ApplyEdges( $\text{Up}(\text{next})$ )  $\wedge$   $\neg$ reachable;
8   | layers.push(next);
9   | if (next  $\wedge$  enc( $l_T$ ))  $\neq$  false then
10    |   | return ExtractTrace(layers);
11 return Not reachable;

```

4.3 Model-Checking Algorithm

Algorithm 3 shows how to check the reachability of a target location given an abstract domain. The list `layers` contains, at position i , the set of states that are reachable in i steps. The function `ApplyEdges` computes the disjunction of immediate successors by all edges. It consists in looping over all edges $e = (l_1, g, R, l_2)$, and gathering the following image by e :

$$\text{enc}(\ell_2) \wedge \text{Reset}_{r_k}(\text{Reset}_{r_{k-1}}(\dots(\text{Reset}_{r_1}(((\exists \mathcal{B}. A(\mathcal{B}, \mathcal{P}) \wedge \text{enc}(\ell_1)) \wedge \alpha_{\mathcal{D}}(g)))))),$$

where $R = \{r_1, \dots, r_k\}$. We thus use a partitioned transition relation and do not compute the monolithic transition relation.

When the target location is found to be reachable, `ExtractTrace(layers)` returns a trace reaching the target location. This is standard and can be done by computing backwards from the last element of `layers`, by finding which edge can be applied to reach the current state. Since both reset and time successor operations are defined using relations, predecessors in our abstract system can be easily computed using the operator `preR`. As it is standard, we omit the precise definition of this function (the reader can refer to the implementation) but assume that it returns a trace of the form $A_1 \xrightarrow{\sigma_1} A_2 \xrightarrow{\sigma_2} \dots \xrightarrow{\sigma_{n-1}} A_n$, where the $A_i(\mathcal{B}, \mathcal{P})$ are minterms and the σ_i belong to the trace alphabet $\Sigma = \{\text{up}, r_\emptyset\} \cup \{r(x)\}_{x \in \mathcal{C}}$, with the following meaning:

- if $A_i \xrightarrow{\text{up}} A_{i+1}$ then $A_{i+1} = \text{Up}(A_i)$;
- if $A_i \xrightarrow{r_\emptyset} A_{i+1}$ then $A_{i+1} = A_i$;
- if $A_i \xrightarrow{r(x)} A_{i+1}$ then $A_{i+1} = \text{Reset}_x(A_i)$.

The feasibility of such a trace is easily checked using DBMs.

The overall algorithm then follows a classical CEGAR scheme. We initialize \mathcal{D} by adding the clock constraints that appear syntactically in \mathcal{A} , which is often

a good heuristic. We run the reachability check of Algorithm 3. If no trace is found, then the target location is not reachable. If a trace is found, then we check for feasibility. If it is feasible, then the counterexample is confirmed. Otherwise, the trace is spurious and we run the refinement procedure described in the next subsection, and repeat the analysis.

4.4 Abstraction Refinement

Since we initialize \mathcal{D} with all clock constraints appearing in guards, we can assume that all guards are represented exactly in the considered abstractions. Note that the algorithm can be easily extended to the general case; but this simplifies the presentation.

The abstract transition relation we use is not the most precise abstraction of the concrete transition relation. Therefore, it is possible to have abstract transitions $A_1 \xrightarrow{a} A_2$ for some action a while no concrete transition exists between $\llbracket A_1 \rrbracket$ and $\llbracket A_2 \rrbracket$. This requires care and is not a direct application of the standard refinement technique from [11]. A second difficulty is due to incomplete reduction of the predicates using $\text{reduce}_{\mathcal{D}}^2$. In fact, some reachable states in our abstract model will be unsatisfiable. Let us explain how we refine the abstraction in each of these cases.

Consider an algorithm interp which returns an interpolant of given zones Z_1, Z_2 . In what follows, by the *refinement of \mathcal{D} by $\text{interp}(Z_1, Z_2)$* , we mean the domain \mathcal{D}' obtained by adding (k, \prec) to $\mathcal{D}_{x,y}$ for all constraints $x - y \prec k$ of $\text{interp}(Z_1, Z_2)$. Observe that $\alpha_{\mathcal{D}'}(Z_1) \cap \alpha_{\mathcal{D}'}(Z_2) = \emptyset$ in this case.

We define concrete successor and predecessor operations for the actions in Σ . For each $a \in \Sigma$, let Pre_a^c denote the concrete predecessor operation on zones defined straightforwardly, and similarly for Post_a^c .

Consider domain \mathcal{D} and the induced abstraction function $\alpha_{\mathcal{D}}$. Assume that we are given a spurious trace $\pi = A_1 \xrightarrow{\sigma_1} A_2 \xrightarrow{\sigma_1} \dots \xrightarrow{\sigma_{n-1}} A_n$. Let $B_1 \dots B_n$ be the sequence of concrete states visited along π in \mathcal{A} , that is, B_1 is the concrete initial state, and for all $2 \leq i \leq n$, let $B_i = \text{Post}_{\pi_{i-1}}^c(B_{i-1})$. This sequence can be computed using DBMs.

The trace is *realizable* if $B_n \neq \emptyset$, in which case the counterexample is confirmed. Otherwise it is *spurious*. We show how to refine the abstraction to eliminate a spurious trace π .

Let i_0 be the maximal index such that $B_{i_0} \neq \emptyset$. There are three possible reasons explaining why B_{i_0+1} is empty:

1. first, if the abstract successor A_{i_0+1} is unsatisfiable, that is, if it contains contradictory predicates; in this case, $\llbracket A_{i_0+1} \rrbracket = \emptyset$, and the abstraction is refined by Lemma 8 to eliminate this case by strengthening $\text{reduce}_{\mathcal{D}}^k$.
2. if there are predecessors of A_{i_0+1} inside A_{i_0} but none of them are in B_{i_0} , i.e., $\text{Pre}_{\pi_{i_0}}^c(\llbracket A_{i_0+1} \rrbracket) \cap \llbracket A_{i_0} \rrbracket \neq \emptyset$; in this case, we refine the domain by separating these predecessors from the rest of A_{i_0} using $\text{interp}(\text{Pre}_{\pi_{i_0}}^c(\llbracket A_{i_0+1} \rrbracket), B_{i_0-1})$, as in [11].

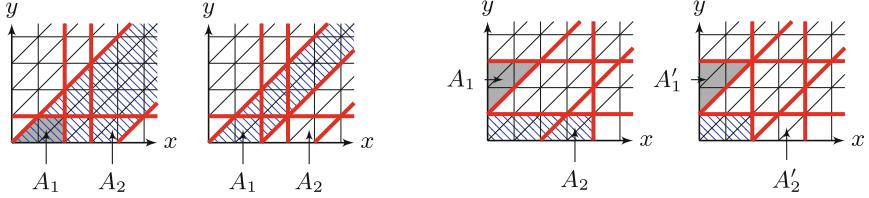
3. otherwise, there are no predecessors of A_{i_0+1} inside A_{i_0} : we refine the abstraction according to the type of the transition from step i_0 to $i_0 + 1$:
- if $\pi_{i_0} = \text{up}$: refine \mathcal{D} by $\text{interp}(\llbracket A_{i_0} \rrbracket \uparrow, \llbracket A_{i_0+1} \rrbracket \downarrow)$.
 - if $\pi_{i_0} = r(x)$: refine \mathcal{D} by $\text{interp}(\text{Free}_x(\llbracket A_{i_0} \rrbracket), \text{Free}_x(\llbracket A_{i_0+1} \rrbracket))$.

Note that the case $\pi_{i_0} = r_\emptyset$ is not possible since this induces the identity function both in the abstract and concrete systems.

Given abstraction $\alpha_{\mathcal{D}}$ and spurious trace π , let $\text{refine}(\alpha_{\mathcal{D}}, \pi)$ denote the refined abstraction $\alpha'_{\mathcal{D}'}$ obtained as described above.

The following two lemmas justify the two subcases of the third case above. They prove that the detected spurious transition disappears after refinement. The reset and up operations depend on the abstraction, so we make this dependence explicit below by using superscripts, as in Reset_x^α and Up^α , in order to distinguish the operations before and after a refinement.

Lemma 12. Consider $(A_1, A_2) \in \text{Up}^\alpha$ with $\llbracket A_1 \rrbracket \uparrow \cap \llbracket A_2 \rrbracket = \emptyset$. Then $\llbracket A_1 \rrbracket \uparrow \cap \llbracket A_2 \rrbracket \downarrow = \emptyset$. Moreover, if α' is obtained by refinement of α by $\text{interp}(\llbracket A_1 \rrbracket \uparrow, \llbracket A_2 \rrbracket \downarrow)$, then for all $(A'_1, A'_2) \in \text{Up}^{\alpha'}$, $\llbracket A'_1 \rrbracket \subseteq \llbracket A_1 \rrbracket$ implies $\llbracket A'_2 \rrbracket \cap \llbracket A_2 \rrbracket = \emptyset$.



(a) Refinement for the time successors operation. The interpolant that separates $\llbracket A_1 \rrbracket \uparrow$ from $\llbracket A_2 \rrbracket \downarrow$ contains the constraint $x = y + 2$. When this is added to the abstract domain, the set A'_2 (which is A_2 in the new abstraction) is no longer reachable by the time successors operation.

(b) Refinement for the reset operation. The interpolant that separates $\text{Free}_y(A_1)$ from $\text{Free}_y(A_2)$ contains the constraint $x < 2$. When this is added to the abstract domain, the set A'_2 (which is A_2 in the new abstraction) is no longer reachable by the reset operation.

Lemma 13. Consider $x \in \mathcal{C}$, and $(A_1, A_2) \in \text{Reset}_x^\alpha$ such that $\llbracket A_1 \rrbracket[x \leftarrow 0] \cap \llbracket A_2 \rrbracket = \emptyset$. Then $\text{Free}_x(\llbracket A_1 \rrbracket) \cap \text{Free}_x(\llbracket A_2 \rrbracket) = \emptyset$. Moreover, if α' is obtained by refinement of α by $\text{interp}(\text{Free}_x(\llbracket A_1 \rrbracket), \text{Free}_x(\llbracket A_2 \rrbracket))$, then for all $(A'_1, A'_2) \in \text{Reset}_x^{\alpha'}$ with $\llbracket A'_1 \rrbracket \subseteq \llbracket A_1 \rrbracket$, we have $\llbracket A'_2 \rrbracket \cap \llbracket A_2 \rrbracket = \emptyset$.

5 Experiments

We implemented both algorithms. The symbolic version was implemented in OCaml using the CUDD library²; the explicit version was implemented in C++ within an existing model checker using Uppaal DBM library. Both prototypes

² <http://vlsi.colorado.edu/~fabio/>.

take as input networks of timed automata with invariants, discrete variables, urgent and committed locations. The presented algorithms are adapted to these features without difficulty.

We evaluated our algorithms on three classes of benchmarks we believe are significant. We compare the performance of the algorithm with that of Uppaal [7] which is based on zones, as well as the BDD-based model checker engine of PAT [25]. We were unable to compare with RED [30] which is not maintained anymore and not open source, and with which we failed to obtain correct results. The tool used in [16] was not available either. We thus only provide a comparison here with two well-maintained tools.

Two of our benchmarks are variants of schedulability-analysis problems where task execution times depend on the internal states of executed processes, so that an analysis of the state space is necessary to obtain a precise answer.

Monoprocess Scheduling Analysis. In this variant, a single process sequentially executes tasks on a single machine, and the execution time of each cycle depends on the state of the process. The goal is to determine a bound on the maximum execution time of a single cycle. This depends on the semantics of the process since the bound depends on the reachable states.

More precisely, we built a set of benchmarks where the processes are defined by synchronous circuit models taken from the Synthesis Competition (<http://www.syntcomp.org>). We assume that each latch of the circuit is associated with a resource, and changing the state of the resource takes some amount of time. So a subset of the latches have clocks associated with them, which measure the time elapsed since the latest value change (latest moment when the value changed from 0 to 1, or from 1 to 0). We provide two time positive bounds ℓ_0 and ℓ_1 for each latch, which determine the execution time as follows: if the value of latch ℓ changes from 0 to 1 (resp. from 1 to 0), then the execution time of the present cycle cannot be less than ℓ_1 (resp. ℓ_0). The execution time of the step is then the minimum that satisfies these constraints.

Multi-process Stateful Scheduling Analysis. In this variant, three processes are scheduled on two machines with a round-robin policy. Processes schedule tasks one after the other without any delay. As in the previous benchmarks, a process executing a task (on any machine) corresponds to a step of the synchronous circuit model. Each task is described by a tuple (C_1, C_2, D) which defines the minimum and maximum execution times, and the relative deadline. When a task finishes, the next task arrives immediately. The values in the tuple depend on the state of the process. The goal is to check the absence of any deadline miss. Processes are also instantiated with AIG circuits from <http://www.syntcomp.org>.

Asynchronous Computation. We consider an asynchronous network of “threshold gates”, defined as follows: each gate is characterized by a tuple $(n, \theta, [l, u])$ where n is the number of inputs, $0 \leq \theta \leq n$ is the threshold, and $l \leq u$ are lower and upper bounds on activation time. Each gate has an output which is initially undefined. The gate becomes active during the time period $[l, u]$.

During this time, if all inputs are defined, and if at least θ of the inputs have value 1, then it sets its output to 1. At the end of the time period, it becomes deactivated and the output becomes undefined again, until the next period, which starts l time units after the deactivation. The goal is to check whether the given gate can output 1 within a given time bound T .

Results. Figure 3 displays the results of our experiments. All algorithms were given 8 GB of memory and a timeout of 30 min, and the experiments were run on laptop with an Intel i7@3.2 Ghz processor running Linux. The symbolic algorithm performs best among all on the monoprocess and multiprocess scheduling benchmarks. Uppaal is the second best, but does not solve as many benchmarks as our algorithm. Our enumerative algorithm quickly fails on these benchmarks, often running out of memory. On asynchronous computation benchmarks, our enumerative algorithm performs remarkably well, beating all other algorithms. We ran our tools on the CSMA/CD benchmarks (with 3 to 12 processes); Uppaal performs the best but our enumerative algorithm is slightly behind. The symbolic algorithm does not scale, while PAT fails to terminate in all cases.

The tool used for the symbolic algorithm is open source and can be found at <https://github.com/osankur/symrob> along with all the benchmarks.

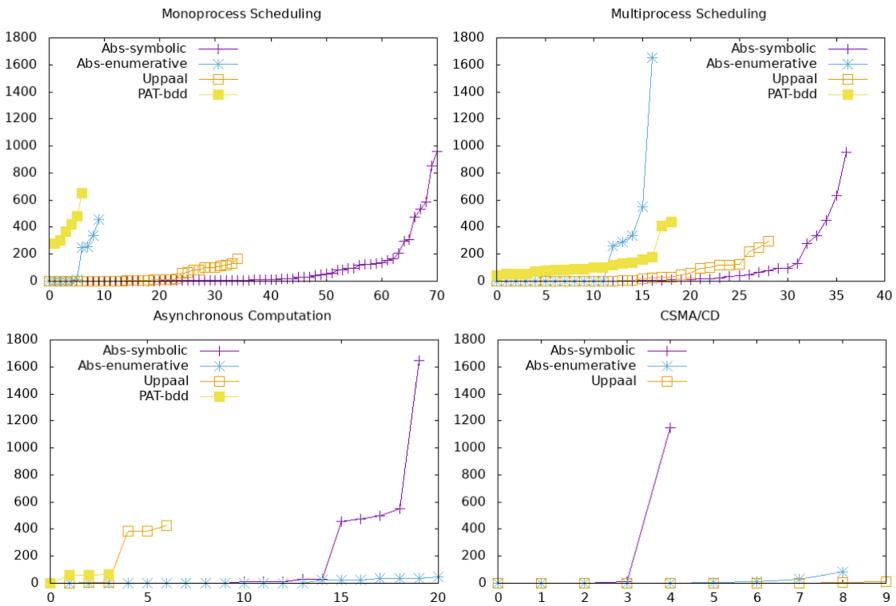


Fig. 3. Comparison of our enumerative and symbolic algorithms (referred to as Abs-enumerative and Abs-symbolic) with Uppaal and PAT. Each figure is a cactus plot for the set of benchmarks: a point (X, Y) means X benchmarks were solved within time bound Y .

6 Conclusion and Future Work

There are several ways to improve the algorithm. Since the choice of interpolants determines the abstraction function and the number of refinements, we assumed that taking the minimal interpolant should be preferable as it should keep the abstractions as coarse as possible. But it might be better to predict which interpolant is the most adapted for the rest of the computation in order to limit future refinements. The number of refinement also depends on the search order, and although it has already been studied in [23], it could be interesting to study it in this case. Generally speaking, it is worth noting that we currently cannot predict which (variant of) our algorithms is better suited for which model.

Several extensions of our algorithms could be developed, *e.g.* combining our algorithms with other methods based on finer abstractions as in [22], integrating predicate abstraction on discrete variables, or developing SAT-based versions of our algorithms.

References

1. Althaus, E., et al.: Verification of linear hybrid systems with large discrete state-spaces using counterexample-guided abstraction refinement. *Sci. Comput. Program.* **1**(48), 123–160 (2017)
2. Alur, R., Courcoubetis, C., Dill, D.L.: Model-checking in dense real-time. *Inf. Comput.* **104**(1), 2–34 (1993)
3. Alur, R., Dill, D.: Automata for modeling real-time systems. In: Paterson, M.S. (ed.) ICALP 1990. LNCS, vol. 443, pp. 322–335. Springer, Heidelberg (1990). <https://doi.org/10.1007/BFb0032042>
4. Baier, Ch., Katoen, J.-P.: Principles of Model-Checking. MIT Press, Cambridge (2008)
5. Ball, T., Rajamani, S.K.: The SLAM toolkit. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 260–264. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44585-4_25
6. 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.) TACAS 2004. LNCS, vol. 2988, pp. 312–326. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24730-2_25
7. Behrmann, G.: Uppaal 4.0. In: Proceedings of the 3rd International Conference on Quantitative Evaluation of Systems (QEST 2006), pp. 125–126. IEEE Computer Society Press, September 2006
8. Bengtsson, J., Yi, W.: Timed automata: semantics, algorithms and tools. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) ACPN 2003. LNCS, vol. 3098, pp. 87–124. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27755-2_3
9. Berthomieu, B., Menasche, M.: An enumerative approach for analyzing time Petri nets. In: Mason, R.E.A. (eds.) Information Processing—Proceedings of the 9th IFIP World Computer Congress (WCC 1983), pp. 41–46. North-Holland/IFIP, September 1983
10. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Kozen, D. (ed.) Logic of Programs 1981. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1982). <https://doi.org/10.1007/BFb0025774>

11. 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)
12. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (2000)
13. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Conference Record of the 4th ACM Symposium on Principles of Programming Languages (POPL 1977), pp. 238–252. ACM Press, January 1977
14. Dierks, H., Kupferschmid, S., Larsen, K.G.: Automatic abstraction refinement for timed automata. In: Raskin, J.-F., Thiagarajan, P.S. (eds.) FORMATS 2007. LNCS, vol. 4763, pp. 114–129. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75454-1_10
15. Dill, D.L.: Timing assumptions and verification of finite-state concurrent systems. In: Sifakis, J. (ed.) CAV 1989. LNCS, vol. 407, pp. 197–212. Springer, Heidelberg (1990). https://doi.org/10.1007/3-540-52148-8_17
16. Ehlers, R., Fass, D., Gerke, M., Peter, H.-J.: Fully symbolic timed model checking using constraint matrix diagrams. In: Proceedings of the 31st IEEE Symposium on Real-Time Systems (RTSS 2010), pp. 360–371. IEEE Computer Society Press, November 2010
17. Ehlers, R., Mattmüller, R., Peter, H.-J.: Combining symbolic representations for solving timed games. In: Chatterjee, K., Henzinger, T.A. (eds.) FORMATS 2010. LNCS, vol. 6246, pp. 107–121. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15297-9_10
18. He, F., Zhu, H., Hung, W.N.N., Song, X., Gu, M.: Compositional abstraction refinement for timed systems. In: 2010 4th IEEE International Symposium on Theoretical Aspects of Software Engineering, pp. 168–176, August 2010
19. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Conference Record of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2002). ACM Press, January 2002. ACM SIGPLAN Notices **37**(1), 58–70
20. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Software verification with BLAST. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 235–239. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-44829-2_17
21. Herbребеаu, F., Kini, D., Srivathsan, B., Walukiewicz, I.: Using non-convex approximations for efficient analysis of timed automata. In: Chakraborty, S., Kumar, A. (eds.) Proceedings of the 31st Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2011), volume 13 of Leibniz International Proceedings in Informatics, pp. 78–89. Leibniz-Zentrum für Informatik, December 2011
22. Herbребеаu, F., Srivathsan, B., Walukiewicz, I.: Lazy abstractions for timed automata. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 990–1005. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_71
23. Herbребеаu, F., Tran, T.-T.: Improving search order for reachability testing in timed automata. In: Sankaranarayanan, S., Vicario, E. (eds.) FORMATS 2015. LNCS, vol. 9268, pp. 124–139. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-22975-1_9
24. McMillan, K.L.: Symbolic model checking—an approach to the state explosion problem. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA (1993)

25. Nguyen, T.K., Sun, J., Liu, Y., Dong, J.S., Liu, Y.: Improved BDD-based discrete analysis of timed systems. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 326–340. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32759-9_28
26. Pnueli, A.: The temporal logic of programs. In: Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS 1977), pp. 46–57. IEEE Computer Society Press, October–November 1977
27. Roussanaly, V., Sankur, O., Markey, N.: Abstraction refinement algorithms for timed automata. Technical report [arXiv:1905.07365](https://arxiv.org/abs/1905.07365) arXiv, May 2019
28. Seshia, S.A., Bryant, R.E.: Unbounded, fully symbolic model checking of timed automata using boolean methods. In: Hun Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 154–166. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45069-6_16
29. Tóth, T., Majzik, I.: Lazy reachability checking for timed automata using interpolants. In: Abate, A., Geeraerts, G. (eds.) FORMATS 2017. LNCS, vol. 10419, pp. 264–280. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-65765-3_15
30. Wang, F.: Symbolic verification of complex real-time systems with clock-restriction diagram. In: Kim, M., Chin, B., Kang, S., Lee, D. (eds.) Proceedings of the 21st IFIP TC6/WG6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE 2001), volume 197 of IFIP Conference Proceedings, pp. 235–250. Chapman & Hall, August 2001
31. Wang, W., Jiao, L.: Trace abstraction refinement for timed automata. In: Cassez, F., Raskin, J.-F. (eds.) ATVA 2014. LNCS, vol. 8837, pp. 396–410. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11936-6_28

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Fast Algorithms for Handling Diagonal Constraints in Timed Automata

Paul Gastin¹ Sayan Mukherjee² , and B. Srivathsan²

¹ LSV, ENS Paris-Saclay, CNRS, Université Paris-Saclay, Cachan, France
paul.gastin@lsv.fr

² Chennai Mathematical Institute, Chennai, India
{sayanm,sri}@cmi.ac.in

Abstract. A popular method for solving reachability in timed automata proceeds by enumerating reachable sets of valuations represented as zones. A naïve enumeration of zones does not terminate. Various termination mechanisms have been studied over the years. Coming up with efficient termination mechanisms has been remarkably more challenging when the automaton has diagonal constraints in guards.

In this paper, we propose a new termination mechanism for timed automata with diagonal constraints based on a new simulation relation between zones. Experiments with an implementation of this simulation show significant gains over existing methods.

Keywords: Timed automata · Diagonal constraints · Reachability · Zones · Simulations

1 Introduction

Timed automata have emerged as a popular model for systems with real-time constraints [2]. Timed automata are finite automata extended with real-valued variables called *clocks*. All clocks are assumed to start at 0, and increase at the same rate. Transitions of the automaton can make use of these clocks to disallow behaviours which violate timing constraints. This is achieved by making use of *guards* which are constraints of the form $x \leq 5$, $x - y \geq 3$, $y > 7$, etc. where x, y are clocks. A transition guarded by $x \leq 5$ says that it can be fired only when the value of clock x is ≤ 5 . Another important feature is the *reset* of clocks in transitions. Each transition can specify a subset of clocks whose values become 0 once the transition is fired. The combination of guards and resets allows to track timing distance between events. A basic question that forms the core of timed automata technology is *reachability*: given a timed automaton, does there

This work is supported by UMI Relax. The first author is partly supported by ANR project TickTac (ANR-18-CE40-0015) and third author by CEFIPRA project IoT TTA (Indo-French program in ICST-DST/CNRS ref. 2016-01). The second and third authors are partly supported by Infosys Foundation (India) and Tata Consultancy Services - Innovation Labs (Pune, India).

© The Author(s) 2019

I. Dillig and S. Tasiran (Eds.): CAV 2019, LNCS 11561, pp. 41–59, 2019.

https://doi.org/10.1007/978-3-030-25540-4_3

exist an execution from its initial state to a final state. This question is known to be decidable [2]. Various algorithms for this problem have been studied over the years and have been implemented in tools [6, 21, 26, 28, 31, 32].

Since the clocks are real valued variables, the space of configurations of a timed automaton (consisting of a state and a valuation of the clocks) is infinite and an explicit enumeration is not possible. The earliest solution to reachability was to partition this space into a finite number of *regions* and build a region graph that provides a finite abstraction of the behaviour of the timed automaton [2]. However, this solution was not practical. Subsequent works introduced the use of *zones* [14]. Zones are special sets of clock valuations with efficient data structures and manipulation algorithms [6]. Within zone based algorithms, there is a division: forward analysis versus backward analysis. The current industry strength tool UPPAAL [28] implements a forward analysis approach, as this works better in the presence of other discrete data structures used in UPPAAL models [9]. We focus on this forward analysis approach using zones in this paper.

The forward analysis of a timed automaton essentially enumerates sets of reachable configurations stored as zones. Some extra care needs to be taken for this enumeration to terminate. Traditional development of timed automata made use of *extrapolation* operators over zones to ensure termination. These are functions which map a zone to a bigger zone. Importantly, the range of these functions is finite. The goal was to come up with extrapolation operators which are sound: adding these extra valuations should not lead to new behaviours. This is where the role of *simulations* between configurations was studied and extrapolation operators based on such simulations were devised [14]. A certain extrapolation operation, which is now known as Extra_M [5] was proposed and reachability using Extra_M was implemented in tools [14].

A seminal paper by Bouyer [9] revealed that Extra_M is not correct in the presence of *diagonal constraints* in guards. These are constraints of the form $x - y \triangleleft c$ where \triangleleft is either $<$ or \leq , and c is an integer. Moreover, it was proved that no such extrapolation operation would be correct when there are diagonal constraints present. It was shown that for automata without diagonal constraints (henceforth referred to as diagonal-free automata), the extrapolation works. After this result, developments in timed automata reachability focussed on the class of diagonal-free automata [4, 5, 23, 24], and diagonal constraints were mostly sidelined. All these developments have led to quite efficient algorithms for diagonal-free timed automata.

Diagonal constraints are a useful modeling feature and occur naturally in certain problems, especially scheduling [3, 17, 20, 27] and logic-automata translations [16, 25], also in [29]. It is however known that they do not add any expressive power: every timed automaton can be converted into a diagonal-free timed automaton [7]. This conversion suffers from an exponential blowup, which was later shown to be unavoidable: diagonal constraints could potentially give exponentially more succinct models [10]. Therefore, a good forward analysis algorithm that works directly on a timed automaton with diagonal constraints would be handy. This is the subject of this paper.

Related Work. The first attempt at such an algorithm was to split the (extrapolated) zones with respect to the diagonal constraints present in the automaton [6]. This gave a correct procedure, but since zones are split, an enumeration starts from each small zone leading to an exponential blow-up in the number of visited zones. A second attempt was to do a more refined conversion into a diagonal free automaton by detecting “relevant” diagonals [13, 30] in an iterative manner. In order to do this, special data structures storing sets of sets of diagonal constraints were utilized. In [18] we extended the works [5] and [23] on diagonal-free automata to the case of diagonal constraints. All the approaches suffer from either a space or time bottleneck and are incomparable to the efficiency and scalability of tools for diagonal-free automata.

Our Contributions. The goal of this paper is to come up with fast algorithms for handling diagonal constraints. Since the extrapolation based approach is a dead end, we work with simulation between zones directly, as in [23] and [18]. We propose a new simulation relation between zones that is correct in the presence of diagonal constraints (Sect. 3). We give an algorithm to test this simulation between zones (Sect. 4). We have incorporated this simulation test in (an older version of) the tool TChecker [21] checking reachability for timed automata, and compared our results with the state-of-the-art tool UPPAAL. Experiments show an encouraging gain, both in the number of zones enumerated and in the time taken by the algorithm, sometimes upto four orders of magnitude (Sect. 6). The main advantage of our approach is that it does not split zones, and furthermore it leverages the optimizations studied for diagonal-free automata.

From a technical point of view, our presentation does not make use of regions and instead works with valuations, zones and simulation relations. We think that this presentation provides a clearer perspective - as a justification of this claim, we extend our simulation to timed automata with general updates of the form $x := c$ and $x := y + d$ in transitions (where x, y are clocks and c, d are constants) in a rather natural manner (Sect. 5). In general, reachability for timed automata with updates is undecidable [12]. Some decidable cases have been proposed for which the algorithms are based on regions. For decidable subclasses containing diagonal constraints, no zone based approach has been studied. Our proposed method includes these classes, and also benefits from zones and standard optimizations studied for diagonal-free automata.

Missing proofs can be found in the full version of this paper [19].

2 Preliminaries

Let \mathbb{N} be the set of natural numbers, $\mathbb{R}_{\geq 0}$ the set of non-negative reals and \mathbb{Z} the set of integers. Let X be a finite set of variables ranging over $\mathbb{R}_{\geq 0}$, called *clocks*. Let $\Phi(X)$ denote the set of constraints φ formed using the following grammar: $\varphi := x \triangleleft c \mid c \triangleleft x \mid x - y \triangleleft d \mid \varphi \wedge \varphi$, where $x, y \in X$, $c \in \mathbb{N}$, $d \in \mathbb{Z}$ and $\triangleleft \in \{<, \leq\}$. Constraints of the form $x \triangleleft c$ and $c \triangleleft x$ are called *non-diagonal constraints* and those of the form $x - y \triangleleft c$ are called *diagonal constraints*. We have adopted a convention that in non-diagonal constraints $x \triangleleft c$ and $c \triangleleft x$, the

constant c is restricted to \mathbb{N} . A *clock valuation* v is a function which maps every clock $x \in X$ to a real number $v(x) \in \mathbb{R}_{\geq 0}$. A valuation is said to satisfy a guard g , written as $v \models g$ if replacing every x in g with $v(x)$ makes the constraint g true. For $\delta \in \mathbb{R}_{\geq 0}$ we write $v + \delta$ for the valuation which maps every x to $v(x) + \delta$. Given a subset of clocks $R \subseteq X$, we write $[R]v$ for the valuation which maps each $x \in R$ to 0 and each $x \notin R$ to $v(x)$.

A *timed automaton* \mathcal{A} is a tuple (Q, X, q_0, T, F) where Q is a finite set of states, X is a finite set of clocks, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is a set of accepting states and $T \in Q \times \Phi(X) \times 2^X \times Q$ is a set of transitions. Each transition $t \in T$ is of the form (q, g, R, q') where q and q' are respectively the source and target states, g is a constraint called the *guard*, and R is a set of clocks which are *reset* in t . We call a timed automaton *diagonal-free* if guards in transitions do not use diagonal constraints.

A *configuration* of \mathcal{A} is a pair (q, v) where $q \in Q$ and v is a valuation. The semantics of a timed automaton is given by a transition system $\mathcal{S}_{\mathcal{A}}$ whose states are the configurations of \mathcal{A} . Transitions in $\mathcal{S}_{\mathcal{A}}$ are of two kinds: *delay* transitions are given by $(q, v) \xrightarrow{\delta} (q, v + \delta)$ for all $\delta \geq 0$, and *action* transitions are given by $(q, v) \xrightarrow{t} (q', v')$ for each $t := (q, g, R, q')$, if $v \models g$ and $v' = [R]v$. We write $\xrightarrow{\delta, t}$ for a sequence of delay δ followed by action t . A run of \mathcal{A} is an alternating sequence of delay-action transitions starting from the initial state q_0 and the initial valuation $\mathbf{0}$ which maps every clock to 0: $(q_0, \mathbf{0}) \xrightarrow{\delta_0, t_0} (q_1, v_1) \xrightarrow{\delta_1, t_1} \cdots (q_n, v_n)$. A run of the above form is said to be accepting if the last state $q_n \in F$. The *reachability problem* for timed automata is the following: given an automaton \mathcal{A} , decide if there exists an accepting run. This problem is known to be PSPACE-complete [2]. Since the semantics $\mathcal{S}_{\mathcal{A}}$ is infinite, solutions to the reachability problem work with a finite abstraction of $\mathcal{S}_{\mathcal{A}}$ that is sound and complete. Before we explain one of the popular solutions to reachability, we state a result which allows to convert every timed automaton into a diagonal-free timed automaton.

Theorem 1. [7] *For every timed automaton \mathcal{A} , there exists a diagonal-free timed automaton \mathcal{A}_{df} s.t. there is a bijection between runs of \mathcal{A} and \mathcal{A}_{df} . The number of states in \mathcal{A}_{df} is $2^d \cdot n$ where d is the number of diagonal constraints and n is the number of states of \mathcal{A} .*

The above theorem allows to solve the reachability of a timed automaton \mathcal{A} by first converting it into the diagonal free automaton \mathcal{A}_{df} and then checking reachability on \mathcal{A}_{df} . However, this conversion comes with a systematic exponential blowup (in terms of the number of diagonal constraints present in \mathcal{A}). It was shown in [10] that such a blowup is unavoidable in general. We will now recall the general algorithm for analyzing timed automata, and then move into specific details which depend on whether the automaton has diagonal constraints or not.

Zones and Simulations. Fix a timed automaton \mathcal{A} with clock set X for the rest of the discussion in this section. As the space of valuations of \mathcal{A} is infinite, algorithms work with sets of valuations called *zones*. A zone is set of clock valuations given by a conjunction of constraints of the form $x - y \triangleleft c$, $x \triangleleft c$ and

$c \triangleleft x$ where $c \in \mathbb{Z}$ and $\triangleleft \in \{<, \leq\}$, for example the solutions of $x - y < 5 \wedge y \leq 10$ is a zone. The transition relation over configurations (q, v) is extended to (q, Z) where Z is a zone. We define the following operations on zones given a guard g and a set of clocks R : time elapse $\overrightarrow{Z} = \{v + \delta \mid v \in Z, \delta \geq 0\}$; guard intersection $Z \wedge g := \{v \mid v \in Z \text{ and } v \models g\}$ and reset $[R]Z := \{[R]v \mid v \in Z\}$. It can be shown that all these operations result in zones. Zones can be efficiently represented and manipulated using Difference Bound Matrices (DBMs) [15].

The *zone graph* $ZG(\mathcal{A})$ of timed automaton \mathcal{A} is a transition system whose nodes are of the form (q, Z) where q is a state of \mathcal{A} and Z is a zone. For each transition $t := (q, g, R, q')$ of \mathcal{A} , and each zone (q, Z) there is a transition $(q, Z) \Rightarrow^t (q', Z')$ where $Z' = \overrightarrow{[R](Z \wedge g)}$. The initial node is (q_0, Z_0) where q_0 is the initial state of \mathcal{A} and $Z_0 = \{\mathbf{0} + \delta \mid \delta \geq 0\}$ is the zone obtained by elapsing an arbitrary delay from the initial valuation. A path in the zone graph is a sequence $(q_0, Z_0) \Rightarrow^{t_0} (q_1, Z_1) \Rightarrow^{t_1} \dots \Rightarrow^{t_{n-1}} (q_n, Z_n)$ starting from the initial node. The path is said to be accepting if q_n is an accepting state. The zone graph is known to be sound and complete for reachability.

Theorem 2. [14] \mathcal{A} has an accepting run iff $ZG(\mathcal{A})$ has an accepting path.

This does not yet give an algorithm as the zone graph $ZG(\mathcal{A})$ is still not finite. Moreover, there are examples of automata for which the reachable part of $ZG(\mathcal{A})$ is also infinite: starting from the initial node, applying the successor computation leads to infinitely many zones. Two different approaches have been studied to get finiteness, both of them based on the usage of *simulation relations*.

A (time-abstract) simulation relation (\preccurlyeq) between configurations of \mathcal{A} is a reflexive and transitive relation such that $(q, v) \preccurlyeq (q', v')$ implies $q = q'$ and (1) for every $\delta \geq 0$, there exists $\delta' \geq 0$ such that $(q, v + \delta) \preccurlyeq (q, v' + \delta')$ and (2) for every transition t of \mathcal{A} , if $(q, v) \xrightarrow{t} (q_1, v_1)$ then $(q, v') \xrightarrow{t} (q_1, v'_1)$ such that $(q_1, v_1) \preccurlyeq (q_1, v'_1)$.

We say $v \preccurlyeq v'$, read as v is simulated by v' if $(q, v) \preccurlyeq (q, v')$ for all states q . The simulation relation can be extended to zones: $Z \preccurlyeq Z'$ if for every $v \in Z$ there exists $v' \in Z'$ such that $v \preccurlyeq v'$. We write $\downarrow Z$ for $\{v \mid \exists v' \in Z \text{ s.t. } v \preccurlyeq v'\}$. The simulation relation \preccurlyeq is said to be finite if the function mapping zones Z to the down sets $\downarrow Z$ has finite range. We now recall a specific simulation relation \preccurlyeq_{LU} [5, 23]. Current algorithms and tools for diagonal-free automata are based on this simulation. The conditions required for $v \preccurlyeq_{LU} v'$ ensure that when all lower bound constraints $c \triangleleft x$ satisfy $c \leq L(x)$ and all upper bound constraints $x \triangleleft c$ satisfy $c \leq U(x)$, whenever v satisfies a constraint, v' will also satisfy it.

Definition 1 (LU-bounds and the relation \preccurlyeq_{LU} [5, 23]). An LU-bounds function is a pair of functions $L : X \mapsto \mathbb{N} \cup \{-\infty\}$ and $U : X \mapsto \mathbb{N} \cup \{-\infty\}$ that map each clock to either a non-negative constant or $-\infty$. Given an LU-bounds function, we define $v \preccurlyeq_{LU} v'$ for valuations v, v' if for every clock $x \in X$:

$$v'(x) < v(x) \text{ implies } L(x) < v'(x) \text{ and } v(x) < v'(x) \text{ implies } U(x) < v(x).$$

Reachability in Diagonal-Free Timed Automata. A natural method to get finiteness of the zone graph is to prune the zone graph computation through simulations $Z \preccurlyeq Z'$: do not explore a node (q, Z) if there is an already visited node (q, Z') such that $Z \preccurlyeq Z'$. Since these simulation tests need to be done often during the zone graph computation, an efficient algorithm for performing this test is crucial. Note that $Z \preccurlyeq Z'$ iff $Z \subseteq \downarrow_{LU} Z'$. However, it is known that the set $\downarrow_{LU} Z'$ is not necessarily a zone (this was proved for $\downarrow_{LU} Z'$ in [5]), and hence no simple zone inclusions are applicable. The first algorithms for timed automata followed a different approach, which we call the *extrapolation* approach. In this approach, whenever a new zone Z is discovered by the algorithm, a new zone $\text{Extra}(Z) (\supseteq Z)$ gets computed and stored in the place of Z .

Reachability Algorithm Using Zone Extrapolation. The input to the algorithm is a timed automaton \mathcal{A} . The algorithm maintains two lists, Passed and Waiting. Initially, the node $(q_0, \text{Extra}(Z_0))$ is added to the Waiting list (recall that (q_0, Z_0) is the initial node of the zone graph $ZG(\mathcal{A})$). Wlog. we assume that q_0 is not accepting. The algorithm repeatedly performs the following steps:

Step 1. If Waiting is empty, then return “ \mathcal{A} has no accepting run”; else pick (and remove) a node (q, Z) from Waiting. Add (q, Z) to Passed.

Step 2. For each transition $t := (q, g, R, q_1)$, compute the successor $(q, Z) \Rightarrow^t (q_1, Z_1)$: if $Z_1 \neq \emptyset$ perform the following operations - if q_1 is accepting, return “ \mathcal{A} has an accepting run”; else compute $\hat{Z}_1 := \text{Extra}(Z_1)$ and check if there exists a node (q_1, Z'_1) in Passed or Waiting such that $\hat{Z}_1 \subseteq Z'_1$: if yes, ignore the node (q_1, \hat{Z}_1) , otherwise add (q_1, \hat{Z}_1) to Waiting.

Several extrapolation operators (Extra_M , Extra_{LU} , Extra_{LU}^+) were introduced in [5]. The function Extra_{LU}^+ has nice properties - (1) $\text{Extra}_{LU}^+(Z) \subseteq \downarrow_{LU} Z$ and (2) $\text{Extra}_{LU}^+(Z)$ is a zone for all Z . These properties give an algorithm that performs only efficient zone operations: successor computations and zone inclusions.

Reachability Algorithm Using Simulations. The initial node (q_0, Z_0) is added to the Waiting list. Wlog. we assume that q_0 is not accepting. The algorithm repeatedly performs the following steps:

Step 1. If Waiting is empty, then return “ \mathcal{A} has no accepting run”; else pick (and remove) a node (q, Z) from Waiting. Add (q, Z) to Passed.

Step 2. For each transition $t := (q, g, R, q_1)$, compute the successor $(q, Z) \Rightarrow^t (q_1, Z_1)$: if $Z_1 \neq \emptyset$ perform the following operations - if q_1 is accepting, return “ \mathcal{A} has an accepting run”; else check if there exists a node (q_1, Z'_1) in Passed or Waiting such that $Z_1 \preccurlyeq Z'_1$: if yes, ignore the node (q_1, Z_1) , otherwise add (q_1, Z_1) to Waiting.

An $\mathcal{O}(|X|^2)$ algorithm for $Z \preccurlyeq_{LU} Z'$ was proposed in [23]. The efficiency of this simulation check makes it well suited for use in practice. Moreover, as $\text{Extra}_{LU}^+(Z) \subseteq \downarrow_{LU} Z$, we expect to get more simulations (and hence quicker termination) through \preccurlyeq_{LU} .

Reachability in the Presence of Diagonal Constraints. The \preceq_{LU} relation is no longer a simulation when diagonal constraints are present. Moreover, it was shown in [9] that no extrapolation operator (along the lines of Extra_{LU}^+) can work in the presence of diagonal constraints. The first option to deal with diagonals is to use Theorem 1 to get a diagonal free automaton and then apply the methods discussed previously. One problem with this is the systematic exponential blowup introduced in the number of states of the resulting automaton. Another problem is to get diagnostic information: counterexamples need to be translated back to the original automaton [6]. Various methods have been studied to circumvent the diagonal free conversion and instead work on the automaton with diagonal constraints directly. We recall the approach used in the state-of-the-art tool UPPAAL below.

Zone Splitting [6]. The paper introducing timed automata gave a notion of equivalence between valuations $v \simeq_M v'$ parameterized by a function M mapping each clock x to the maximum constant M among the guards of the automaton that involve x . This equivalence is a finite simulation for diagonal-free automata. Equivalence classes of \simeq_M are called regions. This was extended to the diagonal case by [6] as: $v \simeq_M^d v'$ if $v \simeq_M v'$ and for all diagonal constraints g present in the automaton, if $v \models g$ then $v' \models g$. The \simeq_M^d relation splits the regions further, such that each region is either entirely included inside g , or entirely outside g for each g . The next step is to use this notion of equivalence in zones. The paper [6] follows the extrapolation approach: to each zone Z , an extrapolation operation $\text{Extra}_M(Z)$ is applied; this adds some valuations which are \simeq_M equivalent to valuations in Z ; then it is further split into multiple zones, so that each small zone is either inside g or outside g for each diagonal constraint g . If d is the number of diagonal constraints present in the automaton, this splitting process can give rise to 2^d zones for each zone Z . From each small zone, the zone graph computation is started. Essentially, the exponential blow-up at the state level which appeared in the diagonal-free conversion now appears in the zone level.

In this paper, we propose a new simulation to handle diagonal constraints. This has two advantages - using this avoids the blow-up in the number of nodes arising due to zone splitting, and the simulation test between zones has an efficient implementation and is significantly quicker than the simulation of [18].

3 A New Simulation Relation

We start with a definition of a relation between timed automata configurations, which in some sense “declares” upfront what we need out of a simulation relation that can be used in a reachability algorithm. As we proceed, we will make its description more concrete and give an effective simulation algorithm between zones, that can be implemented. Fix a clock set X . This generates constraints $\Phi(X)$.

Definition 2 (the relation $\sqsubseteq_\mathcal{G}$). Let \mathcal{G} be a (finite or infinite) set of constraints. We say $v \sqsubseteq_\mathcal{G} v'$ if for all $\varphi \in \mathcal{G}$ and all $\delta \geq 0$, $v + \delta \models \varphi$ implies $v' + \delta \models \varphi$.

Our goal is to utilize the above relation in a simulation (as defined in p. xx) for a timed automaton. Directly from the definition, we get the following lemma which shows that the $\sqsubseteq_{\mathcal{G}}$ relation is preserved under time elapse.

Lemma 1. *If $v \sqsubseteq_{\mathcal{G}} v'$, then $v + \delta \sqsubseteq_{\mathcal{G}} v' + \delta$ for all $\delta \geq 0$.*

The other kind of transformation over valuations is resets. Given sets of guards \mathcal{G}_1 , \mathcal{G} and a set of clocks R , we want to find conditions on \mathcal{G}_1 and \mathcal{G} so that if $v \sqsubseteq_{\mathcal{G}_1} v'$ then $[R]v \sqsubseteq_{\mathcal{G}} [R]v'$. To do this, we need to answer this question: what guarantees should we ensure for v, v' (via \mathcal{G}_1) so that $[R]v \sqsubseteq_{\mathcal{G}} [R]v'$. This motivates the next definition.

Definition 3 (weakest pre-condition of $\sqsubseteq_{\mathcal{G}}$ over resets). *For a constraint φ and a set of clocks R , we define a set of constraints $\text{wp}(\sqsubseteq_{\varphi}, R)$ as follows: when φ is of the form $x \triangleleft c$ or $c \triangleleft x$, then $\text{wp}(\sqsubseteq_{\varphi}, R)$ is empty if $x \in R$ and is $\{\varphi\}$ otherwise; when φ is a diagonal constraint $x - y \triangleleft c$, then $\text{wp}(\sqsubseteq_{\varphi}, R)$ is:*

- $\{x - y \triangleleft c\}$ if $\{x, y\} \cap R = \emptyset$
- $\{x \triangleleft c\}$ if $y \in R$, $x \notin R$ and $c \geq 0$
- $\{-c \triangleleft y\}$ if $x \in R$, $y \notin R$ and $-c \geq 0$
- empty, otherwise.

For a set of guards \mathcal{G} , we define $\text{wp}(\sqsubseteq_{\mathcal{G}}, R) := \bigcup_{\varphi \in \mathcal{G}} \text{wp}(\sqsubseteq_{\varphi}, R)$.

Note that the relation $\sqsubseteq_{\mathcal{G}}$ is parameterized by a set of constraints. Additionally, we desire this set to be finite, so that the relation can be used in an algorithm. We need to first link an automaton \mathcal{A} with such a set of constraints. One way to do it is to take the set of all guards present in the automaton and to close it under weakest pre-conditions with respect to all possible subsets of clocks. A better approach is to consider a set of constraints for each state, as in [4] where the parameters for extrapolation (the maximum constants appearing in guards) are calculated at each state.

Definition 4 (State based guards). *Let $\mathcal{A} = (Q, X, q_0, T, F)$ be a timed automaton. We associate a set of guards $\mathcal{G}(q)$ for each state $q \in Q$, which is the least set of guards (for the coordinate-wise subset inclusion order) such that for every transition (q, g, R, q_1) : the guard g and the set $\text{wp}(\sqsubseteq_{\mathcal{G}(q_1)}, R)$ are present in $\mathcal{G}(q)$. More precisely, $\{\mathcal{G}(q)\}_{q \in Q}$ is the least solution to the following set of equations written for each $q \in Q$:*

$$\mathcal{G}(q) = \bigcup_{(q, g, R, q_1) \in T} \{g\} \cup \text{wp}(\sqsubseteq_{\mathcal{G}(q_1)}, R)$$

All constraints present in the set $\text{wp}(\sqsubseteq_{\mathcal{G}(q_1)}, R)$ contain constants which are already present in $\sqsubseteq_{\mathcal{G}(q_1)}$. The least solution to the above set of equations can therefore be obtained by a fixed point computation which starts with $\mathcal{G}(q)$ set to $\bigcup_{(q, g, R, q_1) \in T} \{g\}$ and then repeatedly updates the weakest-preconditions. Since no new constants are generated in this process, the fixed point computation terminates. We now have the ingredients to define a simulation relation over configurations of a timed automaton with diagonal constraints.

Definition 5 (\mathcal{A} -simulation). Let $\mathcal{A} = (Q, X, q_0, T, F)$ be a timed automaton and let the set of guards $\mathcal{G}(q)$ of Definition 4 be associated to every state $q \in Q$. We define a relation $\preccurlyeq_{\mathcal{A}}$ between configurations of \mathcal{A} as $(q, v) \preccurlyeq_{\mathcal{A}} (q, v')$ if $v \sqsubseteq_{\mathcal{G}(q)} v'$.

Lemma 2. The relation $\preccurlyeq_{\mathcal{A}}$ is a simulation on the configurations of timed automaton \mathcal{A} .

As pointed before, Definition 2 gives a declarative description of the simulation and it is unclear how to work with it algorithmically, even when the set of constraints \mathcal{G} is finite. The main issue is with the $\forall \delta$ quantification, which is not finite. We will first provide a characterization that brings out the fact that this $\forall \delta$ quantification is irrelevant for diagonal constraints (essentially because value of $v(x) - v(y)$ does not change with time elapse). Given a set of constraints \mathcal{G} , let $\mathcal{G}^- \subseteq \mathcal{G}$ be the set of non-diagonal constraints in \mathcal{G} .

Proposition 1. $v \sqsubseteq_{\mathcal{G}} v'$ iff $v \sqsubseteq_{\mathcal{G}^-} v'$ and for all diagonal constraints $\varphi \in \mathcal{G}$, if $v \models \varphi$ then $v' \models \varphi$.

It now amounts to solving the $\forall \delta$ problem for non-diagonals. It turns out that the \preccurlyeq_{LU} simulation achieves this, almost. We will see this in more detail in the next section.

4 Algorithm for $Z \sqsubseteq_{\mathcal{G}} Z'$

Fix a finite set of guards \mathcal{G} . Restating the definition of $\sqsubseteq_{\mathcal{G}}$ extended to zones: $Z \sqsubseteq_{\mathcal{G}} Z'$ if for all $v \in Z$ there exists a $v' \in Z'$ such that $v \sqsubseteq_{\mathcal{G}} v'$. In this section, we will view the characterization of $\sqsubseteq_{\mathcal{G}}$ as in Proposition 1 and give an algorithm to check $Z \sqsubseteq_{\mathcal{G}} Z'$ that uses as an oracle a test $Z \sqsubseteq_{\mathcal{G}^-} Z'$. We discuss the computation of $Z \sqsubseteq_{\mathcal{G}^-} Z'$ later in this section. We start with an observation following from Proposition 1.

Lemma 3. Let $\varphi := x - y \triangleleft c$ be a diagonal constraint in \mathcal{G} . Then $Z \sqsubseteq_{\mathcal{G}} Z'$ if and only if $Z \cap \varphi \sqsubseteq_{\mathcal{G}'} Z' \cap \varphi$ and $Z \cap \neg\varphi \sqsubseteq_{\mathcal{G}'} Z'$ where $\mathcal{G}' = \mathcal{G} \setminus \{\varphi\}$.

If \mathcal{G} has no diagonal constraints, $Z \sqsubseteq_{\mathcal{G}} Z'$ if and only if $Z \sqsubseteq_{\mathcal{G}^-} Z'$.

This leads to the following algorithm consisting of two mutually recursive procedures. This algorithm is essentially an implementation of the above lemma, with two optimizations:

- we start with the non-diagonal check in Line 6 of Algorithm 1 - if this is already violated, then the algorithm returns false;
- suppose $Z \sqsubseteq_{\mathcal{G}^-} Z'$, the next task is to perform the checks in the first statement of Lemma 3 - this is done by Algorithm 2; note however that when Algorithm 2 is called, we already have $Z \sqsubseteq_{\mathcal{G}^-} Z'$, hence $Z \cap \neg\varphi \sqsubseteq_{\mathcal{G}^-} Z'$. Therefore we use an optimization in Line 7 by calling Algorithm 2 directly (as the check in Line 6 of Algorithm 1 will be redundant).

```

1 check  $Z \sqsubseteq_{\mathcal{G}} Z'$ :
2   if  $Z = \emptyset$  :
3     return true
4   if  $Z' = \emptyset$  :
5     return false
6   if  $Z \not\sqsubseteq_{\mathcal{G}-} Z'$  :
7     return false
8   return  $Z \sqsubseteq_{\mathcal{G}}^* Z'$ 

```

Algorithm 1

```

1 check  $Z \sqsubseteq_{\mathcal{G}}^* Z'$ :
2   if  $\mathcal{G}$  does not contain any
        diagonal constraints :
3     return true
4   pick a diagonal constraint
         $\varphi = x - y \triangleleft c$  from  $\mathcal{G}$ 
5    $\mathcal{G}' \leftarrow \mathcal{G} \setminus \{\varphi\}$ 
6   if  $Z \cap \neg\varphi \neq \emptyset$  :
7     if  $Z \cap \neg\varphi \not\sqsubseteq_{\mathcal{G}'}^* Z'$  :
8       return false
9   return  $Z \cap \varphi \sqsubseteq_{\mathcal{G}'} Z' \cap \varphi$ 

```

Algorithm 2

Computing $Z \sqsubseteq_{\mathcal{G}-} Z'$. We will use \preccurlyeq_{LU} to approximate $\sqsubseteq_{\mathcal{G}-}$: in our implementation of the above algorithms, we replace $Z \sqsubseteq_{\mathcal{G}-} Z'$ with $Z \preccurlyeq_{LU} Z'$. This works because for an appropriate choice of LU (explained below), we have $Z \preccurlyeq_{LU(\mathcal{G})} Z' \Rightarrow Z \sqsubseteq_{\mathcal{G}-} Z'$. The converse is not true as the LU bounds functions cannot distinguish between guards with $<$ and \leq comparisons. Therefore, the \preccurlyeq_{LU} simulation does not characterize $v \sqsubseteq_{\mathcal{G}-} v'$ completely. Although we are aware of the (rather technical) modifications to \preccurlyeq_{LU} simulation that are needed for this characterization, we choose to use the existing \preccurlyeq_{LU} directly as it is safe to do so and it has already been implemented in tools. This gives us a finer simulation than $v \sqsubseteq_{\mathcal{G}-} v'$.

Definition 6 (LU-bounds from \mathcal{G}). Let \mathcal{G} be a finite set of constraints. We define $LU(\mathcal{G})$ to denote the pair of functions $L_{\mathcal{G}}$ and $U_{\mathcal{G}}$ defined as follows:

$$L_{\mathcal{G}}(x) = \begin{cases} -\infty & \text{if there is no guard of the form } c \triangleleft x \text{ in } \mathcal{G} \\ \max\{c \mid c \triangleleft x \in \mathcal{G}\} & \text{otherwise} \end{cases}$$

$$U_{\mathcal{G}}(x) = \begin{cases} -\infty & \text{if there is no guard of the form } x \triangleleft c \text{ in } \mathcal{G} \\ \max\{c \mid x \triangleleft c \in \mathcal{G}\} & \text{otherwise} \end{cases}$$

Lemma 4. For every set of constraints \mathcal{G} , $v \preccurlyeq_{LU(\mathcal{G})} v'$ implies $v \sqsubseteq_{\mathcal{G}-} v'$.

The above observations call for the next definition and subsequent lemmas.

Definition 7 (approximating $\sqsubseteq_{\mathcal{G}}$). Let \mathcal{G} be a finite set of constraints. We define a relation $\sqsubseteq_{\mathcal{G}}^{LU}$ as follows: $v \sqsubseteq_{\mathcal{G}}^{LU} v'$ if $v \preccurlyeq_{LU(\mathcal{G})} v'$ and for all diagonal constraints $\varphi \in \mathcal{G}$, if $v \models \varphi$ then $v' \models \varphi$. Similarly, define $\preccurlyeq_{\mathcal{A}}^{LU}$ as $(q, v) \preccurlyeq_{\mathcal{A}}^{LU} (q, v')$ if $v \sqsubseteq_{\mathcal{G}(q)}^{LU} v'$.

Lemma 5. The relation $\preccurlyeq_{\mathcal{A}}^{LU}$ is a finite simulation on the configurations of \mathcal{A} .

The above lemma and the fact that $Z \preccurlyeq_{LU(\mathcal{G})} Z'$ can be checked in $\mathcal{O}(|X|^2)$ [23, 33], imply the following theorem.

Theorem 3. *When using $Z \preccurlyeq_{LU(\mathcal{G})} Z'$ in the place of $Z \sqsubseteq_{\mathcal{G}-} Z'$, the algorithm is correct and it terminates in $\mathcal{O}(2^d \cdot |X|^2)$ where d is the number of diagonal guards in \mathcal{G} .*

From a complexity viewpoint, this algorithm is not efficient since it makes an exponential number of calls in the number of diagonal constraints (in fact this may not be avoidable due to Lemma 6, which follows from the NP-hardness result in [18]). Although the above algorithm does involve many calls, the internal operations involved in each call are simple zone manipulations. Moreover, the preliminary checks (for instance line 6 of Algorithm 1) cut short the number of calls. This is visible in our experiments which are very good, especially with respect to running time, as compared to other methods. A similar hardness was shown for a different simulation in [18], but the implementation there indeed witnessed the hardness, as the time taken by that algorithm was unsatisfactory.

Lemma 6. *Deciding $Z \not\sqsubseteq_{\mathcal{G}}^{LU} Z'$ is NP-complete.*

5 Simulations for Updatable Timed Automata

In the timed automata considered so far, clocks are allowed to be reset to 0 along transitions. We consider in this section more sophisticated transformations to clocks in transitions. These are called *updates*. An update $up : \mathbb{R}_{\geq 0}^{|X|} \mapsto \mathbb{R}^{|X|}$ is a function mapping non-negative $|X|$ -dimensional reals (valuations) v to general $|X|$ -dimensional reals (which may a priori not be valuations as the coordinates may be negative). The syntax of the update function up is given by a set of atomic updates up_x to each $x \in X$, which are of the form $x := c$ or $x := y + d$ where $c \in \mathbb{N}$, $d \in \mathbb{Z}$ and $y \in X$ (possibly equal to x). Note that we want d to be an integer, since we allow for decrementing clocks, and on the other hand $c \in \mathbb{N}$ since we have non-negative clocks. Given a valuation v and an update up , the valuation $up(v)$ is:

$$up(v)(x) := \begin{cases} c & \text{if } up_x \text{ is } x := c \\ v(y) + d & \text{if } up_x \text{ is } x := y + d \end{cases}$$

Note that in general, due to the presence of updates $x := y + d$, the update $up(v)$ may not yield a clock valuation. However, when it does give a valuation, it can be used as a transformation in timed automata transitions. We say $up(v) \geq 0$ if $up(v)(x) \geq 0$ for all clocks $x \in X$.

An *updateable timed automaton (UTA)* $\mathcal{A} = (Q, X, q_0, T, F)$ is an extension of a classic timed automaton with transitions of the form (q, g, up, q') where up is an update. Semantics extend in the natural way: delay transitions remain the same, and for action transitions $t := (q, g, up, q')$ we have $(q, v) \xrightarrow{t} (q', v')$ if $v \models g$, $up(v) \geq 0$, and $v' = up(v)$. We allow the transition only if the update results

in a valuation. The reachability problem for these automata is known to be undecidable in general [12]. Various subclasses with decidable reachability have been discussed in the same paper. Decidability proofs in [12] take the following flavour, for a given automaton \mathcal{A} : (1) divide the space of all valuations into a finite number of equivalence classes called *regions* (2) to build the parameters for the equivalence, derive a set of diophantine equations from the guards of \mathcal{A} ; if they have a solution then construct the quotient graph of the equivalence (called region graph) parameterized by the obtained solution and check reachability on it; if the equations have no solution, output that reachability for \mathcal{A} cannot be answered. Sufficient conditions on the nature of the updates that give a solution to the diophantine equations have been tabulated in [12]. When the automaton is diagonal-free, the “region-equivalence” can be used to build an extrapolation operation which in turn can be used in a reachability algorithm with zones. When the automaton contains diagonals, the region-equivalence is used to only build a region graph - no effective zone based approach has been studied.

We use a similar idea, but we have two fundamental differences: (1) we want to obtain reachability through the use of simulations on zones, and (2) we build equations over sets of guards as in Definition 4. The advantage of this approach is that this allows the use of coarser simulations over zones. Even for automata with diagonal constraints and updates, we get a zone based algorithm, instead of resorting to regions which are not efficient in practice.

The notion of simulations as in p. xx remains the same, now using the semantics of transitions with updates. We will re-use the simulation relation $\sqsubseteq_{\mathcal{G}}$. We need to extend Definition 3 to incorporate updates. We do this below. Here is a notation: for an update function up , we write $up(x)$ to be c if up_x is $x := c$, and $up(x)$ to be $y + c$ if up_x is $x := y + c$.

Definition 8 (weakest pre-condition of $\sqsubseteq_{\mathcal{G}}$ over updates).

Let up be an update.

For a constraint φ of the form $x \triangleleft c$ or $c \triangleleft x$, we define $wp(\sqsubseteq_{\varphi}, up)$ to be respectively $\{up(x) \triangleleft c\}$ or $\{c \triangleleft up(x)\}$ if these resulting constraints are of the form $z \triangleleft d$ or $d \triangleleft z$ with $z \in X$ and $d \geq 0$, otherwise $wp(\sqsubseteq_{\varphi}, up)$ is empty.

For a constraint $\varphi : x - y \triangleleft c$, we define $wp(\sqsubseteq_{\varphi}, up)$ to be $\{up(x) - up(y) \triangleleft c\}$ if this constraint is either a diagonal using different clocks, or it is of the form $z \triangleleft d$ or $d \triangleleft z$ with $d \geq 0$, otherwise $wp(\sqsubseteq_{\varphi}, up)$ is empty.

For a set of guards \mathcal{G} , we define $wp(\sqsubseteq_{\mathcal{G}}, up) := \bigcup_{\varphi \in \mathcal{G}} wp(\sqsubseteq_{\varphi}, up)$.

Some examples: $wp(x \leq 5, x := x + 10)$ is empty, since $up(x)$ is $x + 10$, and the guard $x + 10 \leq 5$ is not satisfiable; $wp(x \leq 5, x := x - 10)$ is $x \leq 15$, $wp(x \leq 5, x := c)$ is empty, $wp(x - y \leq 5, \langle x := z_1, y := z_2 + 10 \rangle)$ will be $z_1 - (z_2 + 10) \leq 5$, giving the constraint $z_1 - z_2 \leq 15$, $wp(x - y \leq 5, \langle x := z + c_1, y := z + c_2 \rangle)$ is empty, $wp(x - y \leq 5, \langle x := c_1, y := z + c_2 \rangle)$ is $c = c_1 - 5 - c_2 \leq z$ if $c \geq 0$ and is empty otherwise.

Definition 9 (State based guards). Let $\mathcal{A} = (Q, X, q_0, T, F)$ be a UTA. We associate a set of constraints $\mathcal{G}(q)$ for each state $q \in Q$, which is the least set of constraints (for the coordinate-wise subset inclusion order) such that for

every transition (q, g, up, q_1) : the guard g and the set $\text{wp}(\sqsubseteq_{\mathcal{G}(q_1)}, up)$ are present in $\mathcal{G}(q)$, and in addition constraints that allow the update to happen are also present in \mathcal{G} . The last condition is given by the weakest precondition of the set of constraints $\{x \geq 0 \mid x \in X\}$. Overall, $\{\mathcal{G}(q)\}_{q \in Q}$ is the least solution to the following set of equations, for each $q \in Q$:

$$\mathcal{G}(q) = \bigcup_{(q, g, up, q_1) \in T} (\{g\} \cup \text{wp}(\sqsubseteq_{\{x \geq 0 \mid x \in X\}}, up) \cup \text{wp}(\sqsubseteq_{\mathcal{G}(q_1)}, up))$$

The least solution $\{\mathcal{G}(q)\}_{q \in Q}$ is said to be finite if each $\mathcal{G}(q)$ is a finite set of constraints.

In contrast to the simple reset case, the above set of equations may not have a finite solution. Consider a self-looping transition: $(q, x \triangleleft c, x := x - 1, q)$. We require $x \triangleleft c \in \mathcal{G}(q)$. Now, $\text{wp}(x \triangleleft c, x := x - 1)$ is $x \triangleleft c + 1$ which should be in $\mathcal{G}(q)$ according to the above equation. Continuing this process, we need to add $x \triangleleft d$ for every natural number $d \geq c$. Indeed this is consistent with the undecidability of reachability when subtraction updates are allowed. We deal with the subject of finite solutions to the above equations later in this section. On the other hand, when the above system does have a solution with finite $\mathcal{G}(q)$ at every q , we can use the \mathcal{A} simulation of Definition 5 and its approximation $\preceq_{\mathcal{A}}^{LU}$ to get an algorithm.

Proposition 2. Let $\mathcal{A} = (Q, X, q_0, T, F)$ be a UTA. Let $\{\mathcal{G}(q)\}_{q \in Q}$ be the least solution to the equations given in Definition 9. Then, the relation $\preceq_{\mathcal{A}}$ is a simulation on the configurations of \mathcal{A} .

Lemma 7. For a UTA \mathcal{A} , assume that the least solution $\{\mathcal{G}(q)\}_{q \in Q}$ to the state-based guards equations is finite. Then the relation $\preceq_{\mathcal{A}}^{LU}$ is a finite simulation on the configurations of \mathcal{A} .

Finite Solution to the State-Based Guards Equations. The least solution to the equations of Definition 9 can be obtained by a standard Kleene iteration for fixed points computation. For each $i \geq 0$ and each state q , define:

$$\begin{aligned} \mathcal{G}^0(q) &= \bigcup_{(q, g, up, q') \in T} \{g\} \cup \text{wp}(\sqsubseteq_{\{x \geq 0 \mid x \in X\}}, up) \\ \mathcal{G}^{i+1}(q) &= \bigcup_{(q, g, up, q') \in T} \mathcal{G}^i(q) \cup \text{wp}(\sqsubseteq_{\mathcal{G}^i(q')}, up) \end{aligned}$$

The iteration stabilizes when there exists a k satisfying $\mathcal{G}^{k+1}(q) = \mathcal{G}^k(q)$ for all q . At stabilization, the values $\mathcal{G}^k(q)$ satisfy the equations of Definition 9, and give the required $\mathcal{G}(q)$. However, as we mentioned earlier, this iteration might not stabilize at any k . We will now develop some observations that will help detect after finitely many steps if the iteration will stabilize or not.

Suppose we colour the set $\mathcal{G}^{i+1}(q)$ to red if either there exists a diagonal constraint $x - y \triangleleft c \in \mathcal{G}^{i+1}(q) \setminus \mathcal{G}^i(q)$ (a new diagonal is added) or there exists a

non-diagonal constraint $x \triangleleft c$ or $c \triangleleft x$ in $\mathcal{G}^{i+1}(q) \setminus \mathcal{G}^i(q)$ such that the constant c is strictly bigger than c' for respectively every non-diagonal $x \triangleleft c'$ or $c' \triangleleft x$ in $\mathcal{G}^i(q)$ (a non-diagonal with a bigger constant is added). If this condition is not applicable, we colour the set $\mathcal{G}^{i+1}(q)$ *green*. The next observations say that the iteration terminates iff we reach a stage where all sets are green. Intuitively, once we reach green, the only constraints that can be added are non-diagonals having smaller (non-negative) constants and hence the procedure terminates.

Lemma 8. *Let $i > 0$. If $\mathcal{G}^i(q)$ is green for all q , then $\mathcal{G}^{i+1}(q)$ is green for all q .*

Lemma 9. *Let $K = 1 + |Q| \cdot |X| \cdot (|X| + 1)$. If there is a state p such that $\mathcal{G}^K(p)$ is red, then there is no i such that $\mathcal{G}^i(q)$ is green for all q .*

As to why the bound $K = 1 + |Q| \cdot |X| \cdot (|X| + 1)$ in the lemma above: a red state at stage i arises due to the addition of a constraint φ_i at state p_i , which in turn depends on a state p_{i-1} marked red at stage $i - 1$ due to constraint φ_{i-1} . If we iterate sufficiently long, we will hit a state p , a sequence of transitions from p to p and a constraint φ such that computing the weakest precondition over this loop will give a new constraint with the same set of clocks as φ but with a different constant. This part can be iterated infinitely often.

Proposition 3. *The least solution of the local constraint equations for a UTA is finite iff $\mathcal{G}^K(q)$ is green for all q and where $K = 1 + |Q| \cdot |X| \cdot (|X| + 1)$.*

Theorem 4. *Let \mathcal{A} be a UTA. It is decidable whether the equations in Definition 9 have a finite solution. When these equations do have a finite solution, zone graph enumeration using $\preceq_{\mathcal{A}}^{LU}$ is a sound, complete and terminating procedure for the reachability problem.*

All decidable classes of [12] can be shown decidable with our approach, by showing stabilization of the $\mathcal{G}(q)$ computation.

Lemma 10. *Reachability is decidable in UTA where: guards are non-diagonals and updates are of the form $x := c$, $x := y$, $x := y + c$ where $c \geq 0$ or, guards include diagonal constraints and updates are of the form $x := c$, $x := y$.*

6 Experiments

We have implemented the reachability algorithm for timed automata with diagonal constraints (and only resets as updates) based on the simulation approach (p. xx) using the $\preceq_{\mathcal{A}}^{LU}$ simulation (Definition 7) for pruning zones. The algorithm for $Z \sqsubseteq_{\mathcal{G}}^{LU} Z'$ comes from Sect. 4. Experiments are reported in Table 1. We take model *Cex* from [8, 30] and *Fischer* from [30]. We are not aware of any other “standard” benchmarks containing diagonal constraints. In addition to these two models, we introduce a new benchmark. This is an extension of the job-shop scheduling using (diagonal-free) timed automata [1]. Here the tasks within a job were logically independent. We add some timing dependency between them

Table 1. Experiments: the column $\#\mathcal{D}$ gives the number of diagonal constraints. Four methods have been reported in the table. First two methods, TChecker with our simulation relation $\sqsubseteq_{\mathcal{G}}^{LU}$ and UPPAAL engine for diagonals, have been run on \mathcal{A} , the automata containing diagonal constraints. Whereas, the third and fourth methods are running diagonal-free engines of UPPAAL and TChecker on \mathcal{A}_{df} , a diagonal-free equivalent of \mathcal{A} . Experiments were run on macOS X with 2.3 GHz Intel core i5 processor, and 8 GB RAM. Time is reported in seconds. We set a timeout of 15 min.

Model	# \mathcal{D}	\mathcal{A} : contains diagonals				\mathcal{A}_{df} : diagonal-free equivalent of \mathcal{A}			
		TChecker + $\sqsubseteq_{\mathcal{G}}^{LU}$		UPPAAL		UPPAAL		TChecker	
		Time	Nodes count	Time	Nodes count	Time	Nodes count	Time	Nodes count
Cex 2	4	0.047	241	0.026	2180	0.005	1039	0.067	1039
Cex 3	6	7.399	7111	111.168	182394	1.028	60982	40.092	60982
Cex 4	8	857.662	185209	Timeout	-	734.543	3447119	Timeout	-
Fischer 4	4	0.032	452	307.836	357687	0.009	1815	0.100	1815
Fischer 5	5	0.257	1842	Timeout	-	0.116	12511	1.856	12511
Fischer 7	7	15.032	26812	Timeout	-	174.560	693603	Timeout	-
Job Shop 3	12	0.420	278	23.093	31711	0.003	845	0.312	845
Job Shop 5	20	285.421	10592	Timeout	-	4.633	179607	150.811	179607

which gets naturally modeled using diagonal constraints. Each model considered above is a product of a number of k timed automata. In the table we write the name of the model and the number k of automata involved in the product. We also report the number of diagonal constraints in each of them.

Experimental Results. We report the results of four methods of handling diagonal constraints, as mentioned in the caption of Table 1. Under each method, we report on the number of zones enumerated and the time taken. The first method gives a huge gain over the second one (upto four orders of magnitude in the number of nodes, and even better for time) and gives a less marked, but still significant, gain over the third and fourth methods. We provide a brief explanation of this phenomenon. The performance of the reachability algorithm is dependent on three factors:

- parameters of extrapolation or simulation: M -simulations which use the maximum constant appearing in the guards, versus the LU -simulations which make a distinction between lower bound guards $c \triangleleft x$ and upper bound guards $x \triangleleft c$ (refer to [5] for the exact definitions of extrapolations based on these parameters, and [23] for simulations based on these parameters); LU -simulations are superior to M -simulations.
- computation of the parameters: global parameters which associate a bound to each clock versus the more local state based parameters as in Definition 4 which associate a set of bounds functions to each state [4]; local bounds are superior to global bounds.
- when diagonal constraints are present, whether zones get split or not: each time a zone gets split, new enumerations start from each of the new nodes; clearly, a no-splitting-of-zones approach is superior to zone splitting.

Algorithm of column 1 uses the superior heuristic in all the three optimizations above. The no-splitting-of-zones was possible thanks to our simulation approach, which temporarily splits zones for checking $Z \sqsubseteq_g^{LU} Z'$, but never starts a new exploration from any of the split nodes. The algorithm of column 2, which is implemented in the current version UPPAAL 4.1 uses the inferior heuristic in all the three above. In particular, it is not clear how the extrapolation approach can avoid the zone splitting in an efficient manner. The superiority of our approach gets amplified (by multiplicative factors) when we consider bigger products with many more diagonals. In the third and fourth methods, we give a diagonal free equivalent of the original model (c.f. Theorem 1) and use the UPPAAL and TChecker engines respectively, for diagonal free timed automata. The UPPAAL diagonal free engine is highly optimized, and makes use of the superior heuristics in the first two optimizations mentioned above (the third heuristic is not applicable now as it is a diagonal free automaton). The third and fourth methods can be considered as a good approximation of the zone splitting approach to diagonal constraints using *LU*-abstractions and local guards.

The second and the third methods are the only possibilities of verifying timed models coming with diagonal constraints in UPPAAL. Both these approaches are in principle prone to a $2^{\#D}$ blowup compared to the first approach, where $\#D$ gives the number of diagonal constraints. The table shows that a good extent of this blowup indeed happens. The UPPAAL diagonal free engine uses “minimal constraint systems” [6] for representing zones, whereas TChecker uses DBMs [15]. This explains why even with the same number of nodes visited, UPPAAL performs better in terms of time. We have not included in the table the comparison with two other works dealing with the same problem: the refined diagonal free conversion [30] and the extension of *LU* simulation for diagonals [18]. However, our results are better than the tables reported in these papers.

7 Conclusion

We have proposed a new algorithm for handling diagonal constraints in timed automata, and extended it to automata with general updates. Our approach is based on a simulation relation between zones. From our preliminary experiments, we can infer that the use of simulations is indispensable in the presence of diagonal constraints as zone-splitting can be avoided. Moreover, the fact that the simulation approach stores the actual zones (as opposed to abstracted zones in the extrapolation approach) has enabled optimizations for diagonal-free automata that work with dynamically changing simulation parameters (*LU*-bounds), which are learnt as and when the zones are expanded [22]. Working with actual zones is also convenient for finding cost-optimal paths in priced timed automata [11]. Investigating these in the presence of diagonal constraints is part of future work. Currently, we have not implemented our approach for updateable timed automata. This will also be part of our future work.

Working directly with a model containing diagonal constraints could be convenient (both during modeling, and during extraction of diagnostic traces) and can also potentially give a smaller automaton to begin with. We believe that our experiments provide hope that diagonal constraints can indeed be used.

References

1. Abdeddaim, Y., Asarin, E., Maler, O.: Scheduling with timed automata. *Theor. Comput. Sci.* **354**(2), 272–300 (2006). <https://doi.org/10.1016/j.tcs.2005.11.018>
2. Alur, R., Dill, D.L.: A theory of timed automata. *Theor. Comput. Sci.* **126**(2), 183–235 (1994). [https://doi.org/10.1016/0304-3975\(94\)90010-8](https://doi.org/10.1016/0304-3975(94)90010-8)
3. Amnell, T., Fersman, E., Mokrushin, L., Pettersson, P., Yi, W.: TIMES b—a tool for modelling and implementation of embedded systems. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 460–464. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-46002-0_32
4. Behrmann, G., Bouyer, P., Fleury, E., Larsen, K.G.: Static guard analysis in timed automata verification. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 254–270. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36577-X_18
5. Behrmann, G., Bouyer, P., Larsen, K.G., Pelánek, R.: Lower and upper bounds in zone-based abstractions of timed automata. *STTT* **8**(3), 204–215 (2006). <https://doi.org/10.1007/s10009-005-0190-0>
6. Bengtsson, J., Yi, W.: Timed automata: semantics, algorithms and tools. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) ACPN 2003. LNCS, vol. 3098, pp. 87–124. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27755-2_3
7. Bérard, B., Petit, A., Diekert, V., Gastin, P.: Characterization of the expressive power of silent transitions in timed automata. *Fundamenta Informaticae* **36**(2,3), 145–182 (1998). <https://doi.org/10.3233/FI-1998-36233>
8. Bouyer, P.: Untameable timed automata!. In: Alt, H., Habib, M. (eds.) STACS 2003. LNCS, vol. 2607, pp. 620–631. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36494-3_54
9. Bouyer, P.: Forward analysis of updatable timed automata. *Form. Methods Syst. Des.* **24**(3), 281–320 (2004). <https://doi.org/10.1023/B:FORM.0000026093.21513.31>
10. Bouyer, P., Chevalier, F.: On conciseness of extensions of timed automata. *J. Autom. Lang. Comb.* **10**(4), 393–405 (2005). <https://doi.org/10.25596/jalc-2005-393>
11. Bouyer, P., Colange, M., Markey, N.: Symbolic optimal reachability in weighted timed automata. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9779, pp. 513–530. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41528-4_28
12. Bouyer, P., Dufourd, C., Fleury, E., Petit, A.: Updatable timed automata. *Theor. Comput. Sci.* **321**(2–3), 291–345 (2004). <https://doi.org/10.1016/j.tcs.2004.04.003>
13. Bouyer, P., Laroussinie, F., Reynier, P.-A.: Diagonal constraints in timed automata: forward analysis of timed systems. In: Pettersson, P., Yi, W. (eds.) FORMATS 2005. LNCS, vol. 3829, pp. 112–126. Springer, Heidelberg (2005). https://doi.org/10.1007/11603009_10
14. Daws, C., Tripakis, S.: Model checking of real-time reachability properties using abstractions. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, pp. 313–329. Springer, Heidelberg (1998). <https://doi.org/10.1007/BFb0054180>

15. Dill, D.L.: Timing assumptions and verification of finite-state concurrent systems. In: Sifakis, J. (ed.) CAV 1989. LNCS, vol. 407, pp. 197–212. Springer, Heidelberg (1990). https://doi.org/10.1007/3-540-52148-8_17
16. Ferrère, T.: The compound interest in relaxing punctuality. In: Havelund, K., Peleska, J., Roscoe, B., de Vink, E. (eds.) FM 2018. LNCS, vol. 10951, pp. 147–164. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-95582-7_9
17. Fersman, E., Pettersson, P., Yi, W.: Timed automata with asynchronous processes: schedulability and decidability. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 67–82. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-46002-0_6
18. Gastin, P., Mukherjee, S., Srivathsan, B.: Reachability in timed automata with diagonal constraints. In: Schewe, S., Zhang, L. (eds.) CONCUR 2018. Leibniz International Proceedings in Informatics (LIPIcs), vol. 118, pp. 28:1–28:17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2018). <https://doi.org/10.4230/LIPIcs.CONCUR.2018.28>
19. Gastin, P., Mukherjee, S., Srivathsan, B.: Fast algorithms for handling diagonal constraints in timed automata. CoRR abs/1904.08590 (2019). <http://arxiv.org/abs/1904.08590>
20. Hatvani, L., David, A., Seceleanu, C., Pettersson, P.: Adaptive task automata with earliest-deadline-first scheduling. In: Proceedings of the 14th International Workshop on Automated Verification of Critical Systems (AVoCS 2014), vol. 70. Electronic Communications of the EASST (2014). <https://doi.org/10.14279/tuj.eceasst.70.975>
21. Herbreteau, F., Point, G.: TChecker, April 2019 <https://github.com/fredher/tchecker> (v02)
22. Herbreteau, F., Srivathsan, B., Walukiewicz, I.: Lazy abstractions for timed automata. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 990–1005. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_71
23. Herbreteau, F., Srivathsan, B., Walukiewicz, I.: Better abstractions for timed automata. Inf. Comput. **251**, 67–90 (2016). <https://doi.org/10.1016/j.ic.2016.07.004>
24. Herbreteau, F., Tran, T.-T.: Improving search order for reachability testing in timed automata. In: Sankaranarayanan, S., Vicario, E. (eds.) FORMATS 2015. LNCS, vol. 9268, pp. 124–139. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-22975-1_9
25. Ho, H.: Revisiting timed logics with automata modalities. In: Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control, HSCC 2019, pp. 67–76. ACM, New York (2019). <https://doi.org/10.1145/3302504.3311818>
26. Kant, G., Laarman, A., Meijer, J., van de Pol, J., Blom, S., van Dijk, T.: LTSmin: high-performance language-independent model checking. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 692–707. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_61
27. Krčál, P., Yi, W.: Decidable and undecidable problems in schedulability analysis using timed automata. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 236–250. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24730-2_20
28. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. STTT **1**(1–2), 134–152 (1997). <https://doi.org/10.1007/s100090050010>

29. Ponge, J., Benatallah, B., Casati, F., Toumani, F.: Analysis and applications of timed service protocols. ACM Trans. Softw. Eng. Methodol. **19**(4), 11:1–11:38 (2010). <https://doi.org/10.1145/1734229.1734230>
30. Reynier, P.A.: Diagonal constraints handled efficiently in UPPAAL. In: Research report LSV-07-02. Laboratoire Spécification et Vérification, ENS Cachan, France (2007)
31. Wang, F.: Efficient verification of timed automata with BDD-like data structures. Int. J. Softw. Tools Technol. Transf. **6**(1), 77–97 (2004). <https://doi.org/10.1007/s10009-003-0135-4>
32. Yovine, S.: Kronos: a verification tool for real-time systems. (Kronos user's manual release 2.2). STTT **1**, 123–133 (1997). <https://doi.org/10.1007/s100090050009>
33. Zhao, J., Li, X., Zheng, G.: A quadratic-time dbm-based successor algorithm for checking timed automata. Inf. Process. Lett. **96**(3), 101–105 (2005). <https://doi.org/10.1016/j.ipl.2005.05.027>

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Safety and Co-safety Comparator Automata for Discounted-Sum Inclusion

Suguman Bansal^(✉) and Moshe Y. Vardi

Rice University, Houston, TX 77005, USA

sugumanb@gmail.com

Abstract. *Discounted-sum inclusion* (DS-inclusion, in short) formalizes the goal of comparing quantitative dimensions of systems such as cost, resource consumption, and the like, when the mode of aggregation for the quantitative dimension is discounted-sum aggregation. *Discounted-sum comparator automata*, or *DS-comparators* in short, are Büchi automata that read two infinite sequences of weights synchronously and relate their discounted-sum. Recent empirical investigations have shown that while DS-comparators enable competitive algorithms for DS-inclusion, they still suffer from the scalability bottleneck of Büchi operations.

Motivated by the connections between discounted-sum and Büchi automata, this paper undertakes an investigation of language-theoretic properties of DS-comparators in order to mitigate the challenges of Büchi DS-comparators to achieve improved scalability of DS-inclusion. Our investigation uncovers that DS-comparators possess safety and co-safety language-theoretic properties. As a result, they enable reductions based on subset construction-based methods as opposed to higher complexity Büchi complementation, yielding tighter worst-case complexity and improved empirical scalability for DS-inclusion.

1 Introduction

The analysis of quantitative dimensions of computing systems such as cost, resource consumption, and distance metrics [6, 10, 28] has been studied thoroughly to design efficient computing systems. Cost-aware program-synthesis [14, 16] and low-cost program-repair [25] have found compelling applications in robotics [24, 29], education [22], and the like. *Quantitative verification* facilitates efficient system design by automatically determining if a system implementation is more efficient than a specification model. Investigations in quantitative verification have demonstrated their high computational complexity and practically intractable [17, 23]. This work addresses practical intractability of quantitative verification.

At the core of quantitative verification lies the problem of *quantitative inclusion* which formalizes the goal of determining which of two given systems is more efficient [17, 23, 31]. In quantitative inclusion, quantitative systems are abstracted as weighted automata [7, 21, 32]. A run in a weighted automaton is associated with a sequence of weights. The quantitative dimension of these runs is determined by the weight of runs, which is computed by taking an aggregate of the

run's weight sequence. Quantitative inclusion can be thought of as the quantitative generalization of (qualitative) language inclusion.

A commonly appearing mode of aggregation is that of *Discounted-sum (DS) aggregation* which captures the intuition that weights incurred in the near future are more significant than those incurred later on [19]. The convergence of DS aggregation for all bounded infinite weight-sequences makes it a preferred mode of aggregation across domains: Reinforcement learning [37], planning under uncertainty [34], and game-theory [33]. This work examines the problem of *Discounted-sum inclusion* or *DS-inclusion* that is quantitative inclusion when *discounted sum* is the mode of aggregation.

In theory, DS-inclusion is PSPACE-complete [12]. Recent algorithmic approaches have tapped into language-theoretic properties of discounted-sum aggregate function [12, 18] to design practical algorithms for DS-inclusion [11, 12]. These algorithms use *DS-comparator automata* (*DS-comparator*, in short) as their main technique, and are *purely* automata-theoretic. While these algorithms outperform other existing approaches for DS-inclusion in runtime [15, 17], even these do not scale well on weighted-automata with more than few hundreds of states [11]. This work contributes novel techniques and algorithms for DS-inclusion to address the scalability challenge of DS-inclusion

An in-depth examination of the DS-comparator based algorithm exposes their scalability bottleneck. DS-comparator is a Büchi automaton that relates the discounted-sum aggregate of two (bounded) weight-sequences A and B by determining the membership of the interleaved pair of sequences (A, B) in the language of the comparator. As a result, DS-comparators reduce DS-inclusion to language inclusion between (non-deterministic) Büchi automaton. In spite of the fact that many techniques have been proposed to solve Büchi language inclusion efficiently in practice [4, 20], none of them can avoid at least an exponential blow-up of $2^{\mathcal{O}(n \log n)}$, for an n -sized input, caused by a direct or indirect involvement of Büchi complementation [36, 40].

This work meets the scalability challenge of DS-inclusion by delving deeper into language-theoretic properties of discounted-sum aggregate functions [18] in order to obtain algorithms for DS-inclusion that render both tighter theoretical complexity and improved scalability. Specifically, we prove that DS-comparators are expressed as *safety automata* or *co-safety automata* [26] (Sect. 3.1), and have compact deterministic constructions (Sect. 3.2). Safety and co-safety automata have the property that their complementation is performed by simpler and lower $2^{\mathcal{O}(n)}$ -complexity subset-construction methods [27]. As a result, they facilitate a procedure for DS-inclusion that uses subset-construction based intermediate steps instead of Büchi complementation, yielding an improvement in theoretical complexity from $2^{\mathcal{O}(n \cdot \log n)}$ to $2^{\mathcal{O}(n)}$. Our subset-construction based procedure has yet another advantage over Büchi complementation as they support efficient on-the-fly implementations, yielding practical scalability as well (Sect. 4).

An empirical evaluation of our prototype tool QuIPFly for the proposed procedure against the prior DS-comparator algorithm and other existing approaches for DS-inclusion shows that QuIPFly outperforms them by orders of magnitude both in runtime and the number of benchmarks solved (Sect. 4).

2 Preliminaries and Related Work

A weight-sequence, finite or infinite, is *bounded* if the absolute value of all of its elements are bounded by a fixed number.

Büchi Automaton: A *Büchi automaton* is a tuple $\mathcal{A} = (S, \Sigma, \delta, s_I, \mathcal{F})$, where S is a finite set of *states*, Σ is a finite *input alphabet*, $\delta \subseteq (S \times \Sigma \times S)$ is the *transition relation*, state $s_I \in S$ is the *initial state*, and $\mathcal{F} \subseteq S$ is the set of *accepting states* [39]. A Büchi automaton is *deterministic* if for all states s and inputs a , $|\{s'|(s, a, s') \in \delta \text{ for some } s'\}| \leq 1$. Otherwise, it is *nondeterministic*. A Büchi automaton is *complete* if for all states s and inputs a , $|\{s'|(s, a, s') \in \delta \text{ for some } s'\}| \geq 1$. For a word $w = w_0 w_1 \dots \in \Sigma^\omega$, a *run* ρ of w is a sequence of states $s_0 s_1 \dots$ s.t. $s_0 = s_I$, and $\tau_i = (s_i, w_i, s_{i+1}) \in \delta$ for all i . Let $\inf(\rho)$ denote the set of states that occur infinitely often in run ρ . A run ρ is an *accepting run* if $\inf(\rho) \cap \mathcal{F} \neq \emptyset$. A word w is an *accepting word* if it has an accepting run. The language of Büchi automaton \mathcal{A} , denoted by $\mathcal{L}(\mathcal{A})$ is the set of all words accepted by \mathcal{A} . By abuse of notation, we write $w \in \mathcal{A}$ and $\rho \in \mathcal{A}$ if w and ρ are an accepting word and an accepting run of \mathcal{A} . Büchi automata are closed under set-theoretic union, intersection, and complementation [39].

Safety and Co-safety Properties: Let $\mathcal{L} \subseteq \Sigma^\omega$ be a language over alphabet Σ . A finite word $w \in \Sigma^*$ is a *bad prefix* for \mathcal{L} if for all infinite words $y \in \Sigma^\omega$, $x \cdot y \notin \mathcal{L}$. A language \mathcal{L} is a *safety language* if every word $w \notin \mathcal{L}$ has a bad prefix for \mathcal{L} . A language \mathcal{L} is a *co-safety language* if its complement language is a safety language [5]. When a safety or co-safety language is an ω -regular language, the Büchi automaton representing it is called a safety or co-safety automaton, respectively [26]. Wlog, safety and co-safety automaton contain a *sink state* from which every outgoing transitions loops back to the sink state and there is a transition on every alphabet symbol. All states except the sink state are accepting in a safety automaton, while only the sink state is accepting in a co-safety automaton. Unlike Büchi complementation, complementation of safety and co-safety automaton is conducted by simpler subset construction with a lower $2^{\mathcal{O}(n)}$ blow-up. The complementation of safety automaton is a co-safety automaton, and vice-versa. Safety automata are closed under intersection, and co-safety automata are closed under union.

Comparator Automaton: For a finite-set of integers Σ , an aggregate function $f : \mathbb{Z}^\omega \rightarrow \mathbb{R}$, and equality or inequality relation $R \in \{<, >, \leq, \geq, =, \neq\}$, the *comparison language for f with relation R* is a language of infinite words over the alphabet $\Sigma \times \Sigma$ that accepts a pair (A, B) iff $f(A) R f(B)$ holds. A *comparator automaton (comparator, in short) for aggregate function f and relation R* is an automaton that accepts the comparison language for f with R [12]. A comparator is said to be *regular* if its automaton is a Büchi automaton.

Weighted Automaton: A *weighted automaton* over infinite words is a tuple $\mathcal{A} = (\mathcal{M}, \gamma, f)$, where $\mathcal{M} = (S, \Sigma, \delta, s_I, S)$ is a complete Büchi automaton

with all states as accepting, $\gamma : \delta \rightarrow \mathbb{N}$ is a *weight function*, and $f : \mathbb{N}^\omega \rightarrow \mathbb{R}$ is the *aggregate function* [17, 31]. *Words* and *runs* in weighted automata are defined as in Büchi automata. The *weight-sequence* of run $\rho = s_0s_1\dots$ of word $w = w_0w_1\dots$ is given by $wt_\rho = n_0n_1n_2\dots$ where $n_i = \gamma(s_i, w_i, s_{i+1})$ for all i . The *weight of a run* ρ , denoted by $f(\rho)$, is given by $f(wt_\rho)$. Here the *weight of a word* $w \in \Sigma^\omega$ in weighted automata is defined as $wt_{\mathcal{A}}(w) = \sup\{f(\rho) | \rho \text{ is a run of } w \text{ in } \mathcal{A}\}$.

Quantitative Inclusion: Let P and Q be weighted automata with the *same* aggregate function. The *strict quantitative inclusion problem*, denoted by $P \subset Q$, asks whether for all words $w \in \Sigma^\omega$, $wt_P(w) < wt_Q(w)$. The *non-strict quantitative inclusion problem*, denoted by $P \subseteq Q$, asks whether for all words $w \in \Sigma^\omega$, $wt_P(w) \leq wt_Q(w)$. *Comparison language or comparator of a quantitative inclusion* problem refer to the comparison language or comparator of the associated aggregate function.

Discounted-sum Inclusion: Let $A = A_0, A_1, \dots$ be a weight sequence, $d > 1$ be a rational number. The *discounted-sum* (DS in short) of A with *integer* discount-factor $d > 1$ is $DS(A, d) = \sum_{i=0}^{\infty} \frac{A_i}{d^i}$. DS-comparison language and DS-comparator with discount-factor $d > 1$ are the comparison language and comparator obtained for the discounted-sum aggregate function with discount-factor $d > 1$, respectively. Strict or non-strict discounted-sum inclusion is strict or non-strict quantitative inclusion with the discounted-sum aggregate function, respectively. For brevity, we abbreviate discounted-sum inclusion to DS-inclusion.

Related Work. The decidability of DS-inclusion is an open problem when the discount-factor $d > 1$ is arbitrary. Recent work has established that DS-inclusion is PSPACE-complete when the discount-factor is an integer [12]. This work investigates algorithmic approaches to DS-inclusion with integer discount-factors.

Two contrasting solution approaches have been identified for DS-inclusion. The first approach is *hybrid* [17]. It separates out the language-theoretic aspects of weighted-automata from the numerical aspects, and solves each separately [15, 17]. More specifically, the hybrid approach solves the language-theoretic aspects by DS-determinization [15] and the numerical aspect is performed by linear programming [8, 9] sequentially. To the best of our knowledge, this procedure cannot be performed in parallel. As a result, this approach must always incur the exponential cost of DS-determinization.

The second approach is *purely-automata theoretic* [12]. This approach uses regular DS-comparator to reduce DS-inclusion to language inclusion between non-deterministic Büchi automata [11, 12]. While the purely automata-theoretic approach scales better than the hybrid approach in runtime [11], its scalability suffers from fundamental algorithmic limitations of Büchi language inclusion. A key ingredient of Büchi language-inclusion is Büchi complementation [36]. Büchi complementation is $2^{\mathcal{O}(n \log n)}$ in the worst-case, and is practically intractable [40]. These limitations also feature in the theoretical complexity and practical performance of DS-inclusion. The complexity of DS-inclusion between weighted

automata P and Q with regular DS-comparator C for integer discount-factor $d > 1$ is $|P| \cdot 2^{\mathcal{O}(|P||Q||C| \cdot \log(|P||Q||C|))}$.

This work improves the worst-case complexity and practical performance of the purely automata theoretic approach for DS-inclusion by a closer investigation of language-theoretic properties of DS-comparators. In particular, we identify that DS-comparator for integer discount-factor form a safety or co-safety automata (depending on the relation R). We show that complementation advantage of safety/co-safety automata not only improves the theoretical complexity of DS-inclusion with integer discount-factor but also facilitate on-the-fly implementations that significantly improve practical performance.

3 DS-inclusion with Integer Discount-Factor

This section covers the core technical contributions of this paper. We uncover novel language-theoretic properties of DS-comparison languages and utilize them to obtain tighter theoretical upper-bound for DS-inclusion with integer discount-factor. Unless mentioned otherwise, the discount-factor is an integer.

In Sect. 3.1 we prove that DS-comparison languages are either safety or co-safety for all rational discount-factors. Since DS-comparison languages are ω -regular for integer discount-factors [12], we obtain that DS-comparators for integer discount-factors form safety or co-safety automata. Next, Sect. 3.2 makes use of newly obtained safety/co-safety properties of DS-comparator to present the first deterministic constructions for DS-comparators. These deterministic construction are compact in the sense that they match their non-deterministic counterparts in number of states [11]. Section 3.3 evaluates the complexity of quantitative inclusion with regular safety/co-safety comparators, and observes that its complexity is lower than the complexity for quantitative inclusion with regular comparators. Finally, since DS-comparators are regular safety/co-safety, our analysis shows that the complexity of DS-inclusion is improved as a consequence of the complexity observed for quantitative-inclusion with regular safety/co-safety comparators.

We begin with formal definitions of safety/co-safety comparison languages and safety/co-safety comparators:

Definition 1 (Safety and co-safety comparison languages). Let Σ be a finite set of integers, $f : \mathbb{Z}^\omega \rightarrow \mathbb{R}$ be an aggregate function, and $R \in \{\leq, <, \geq, >, =, \neq\}$ be a relation. A comparison language L over $\Sigma \times \Sigma$ for aggregate function f and relation R is said to be a safety comparison language (or a co-safety comparison language) if L is a safety language (or a co-safety language).

Definition 2 (Safety and co-safety comparators). Let Σ be a finite set of integers, $f : \mathbb{Z}^\omega \rightarrow \mathbb{R}$ be an aggregate function, and $R \in \{\leq, <, \geq, >, =, \neq\}$ be a relation. A comparator for aggregate function f and relation R is a safety comparator (or co-safety comparator) is the comparison language for f and R is a safety language (or co-safety language).

A safety comparator is *regular* if its language is ω -regular (equivalently, if its automaton is a safety automaton). Likewise, a co-safety comparator is *regular* if its language is ω -regular (equivalently, automaton is a co-safety automaton).

By complementation duality of safety and co-safety languages, comparison language for an aggregate function f for non-strict inequality \leq is safety iff the comparison language for f for strict inequality $<$ is co-safety. Since safety languages and safety automata are closed under intersection, safety comparison languages and regular safety comparator for non-strict inequality renders the same for equality. Similarly, since co-safety languages and co-safety automata are closed under union, co-safety comparison languages and regular co-safety comparators for non-strict inequality render the same for the inequality relation. Therefore, it suffices to examine the comparison language for one relation only.

It is worth noting that for weight-sequences A and B and all relations R , we have that $DS(A, d) R DS(B, d)$ iff $DS(A - B, d) R 0$, where $(A - B)_i = A_i - B_i$ for all $i \geq 0$. Prior work [11] shows that we can define *DS-comparison language* with upper bound μ , discount-factor $d > 1$, and relation R to accept infinite and bounded weight-sequence C over $\{-\mu, \dots, \mu\}$ iff $DS(C, d) R 0$ holds. Similarly, DS-comparator with the same parameters μ , $d > 1$, accepts the DS-comparison language with parameters μ , d and R . We adopt these definitions for DS-comparison languages and DS-comparators

Throughout this section, the concatenation of finite sequence x with finite or infinite sequence y is denoted by $x \cdot y$ in the following.

3.1 DS-comparison Languages and Their Safety/Co-safety Properties

The central result of this section is that DS-comparison languages are safety or co-safety languages for all (integer and non-integer) discount-factors (Theorem 1). In particular, since DS-comparison languages are ω -regular for integer discount-factors [12], this implies that DS-comparators for integer discount-factors form safety or co-safety automata (Corollary 1).

The argument for safety/co-safety of DS-comparison languages depends on the property that the discounted-sum aggregate of all bounded weight-sequences exists for all discount-factors $d > 1$ [35].

Theorem 1. *Let $\mu > 1$ be the upper bound. For rational discount-factor $d > 1$*

1. *DS-comparison languages are safety languages for relations $R \in \{\leq, \geq, =\}$*
2. *DS-comparison language are co-safety languages for relations $R \in \{<, >, \neq\}$.*

Proof (Proof sketch). Due to duality of safety/co-safety languages, it suffices to show that DS-comparison language with \leq is a safety language.

Let DS-comparison language with upper bound μ , rational discount-factor $d > 1$ and relation \leq be denoted by $\mathcal{L}_{\leq}^{\mu, d}$. Suppose that $\mathcal{L}_{\leq}^{\mu, d}$ is not a safety language. Let W be a weight-sequence in the complement of $\mathcal{L}_{\leq}^{\mu, d}$ such that W does not have a bad prefix. Then the following hold: (a). $DS(W, d) > 0$ (b).

For all $i \geq 0$, the i -length prefix $W[i]$ of W can be extended to an infinite and bounded weight-sequence $W[i] \cdot Y^i$ such that $DS(W[i] \cdot Y^i, d) \leq 0$.

Note that $DS(W, d) = DS(W[i], d) + \frac{1}{d^i} \cdot DS(W[i \dots], d)$ where $W[i \dots] = W_i W_{i+1} \dots$ and $DS(W[i], d)$ is the discounted-sum of the finite sequence $W[i]$ i.e. $DS(W[i], d) = \sum_{j=0}^{j=i-1} \frac{W[j]}{d^j}$. Similarly, $DS(W[i] \cdot Y^i, d) = DS(W[i], d) + \frac{1}{d^i} \cdot DS(Y^i, d)$. The contribution of tail sequences $W[i \dots]$ and Y^i to the discounted-sum of W and $W[i] \cdot Y^i$, respectively, diminishes exponentially as the value of i increases. In addition, since W and $W[i] \cdot Y^i$ share a common i -length prefix $W[i]$, their discounted-sum values must converge to each other. The discounted sum of W is fixed and greater than 0, due to convergence there must be a $k \geq 0$ such that $DS(W[k] \cdot Y^k, d) > 0$. Contradiction to (b).

Therefore, DS-comparison language with \leq is a safety language. \square

Semantically this result implies that for a bounded-weight sequence C and rational discount-factor $d > 1$, if $DS(C, d) > 0$ then C must have a finite prefix C_{pre} such that the discounted-sum of the finite prefix is so large that no infinite extension by bounded weight-sequence Y can reduce the discounted-sum of $C_{\text{pre}} \cdot Y$ with the same discount-factor d to zero or below.

Prior work shows that DS-comparison languages are expressed by Büchi automata iff the discount-factor is an integer [13]. Therefore:

Corollary 1. *Let $\mu > 1$ be the upper bound. For integer discount-factor $d > 1$*

1. *DS-comparators are regular safety for relations $R \in \{\leq, \geq, =\}$*
2. *DS-comparators are regular co-safety for relations $R \in \{<, >, \neq\}$*

Lastly, it is worth mentioning that for the same reason [13] DS-comparators for non-integer rational discount-factors do not form safety or co-safety automata.

3.2 Deterministic DS-comparator for Integer Discount-Factor

This section issues deterministic safety/co-safety constructions for DS-comparators with integer discount-factors. This is different from prior works since they supply non-deterministic Büchi constructions only [11, 12]. An outcome of DS-comparators being regular safety/co-safety (Corollary 1) is a proof that DS-comparators permit deterministic Büchi constructions, since non-deterministic and deterministic safety automata (and co-safety automata) have equal expressiveness [26]. Therefore, one way to obtain deterministic Büchi construction for DS-comparators is to determinize the non-deterministic constructions using standard procedures [26, 36]. However, this will result in exponentially larger deterministic constructions. To this end, this section offers direct deterministic safety/co-safety automata constructions for DS-comparator that not only avoid an exponential blow-up but also match their non-deterministic counterparts in number of states (Theorem 3).

Key ideas. Due to duality and closure properties of safety/co-safety automata, we only present the construction of deterministic safety automata for DS-comparator with upper bound μ , integer discount-factor $d > 1$ and relation \leq , denoted by $\mathcal{A}_{\leq}^{\mu,d}$. We proceed by obtaining a *deterministic finite automaton*, (DFA), denoted by $\text{bad}(\mu, d, \leq)$, for the language of bad-prefixes of $\mathcal{A}_{\leq}^{\mu,d}$ (Theorem 2). Trivial modifications to $\text{bad}(\mu, d, \leq)$ will furnish the coveted deterministic safety automata for $\mathcal{A}_{\leq}^{\mu,d}$ (Theorem 3).

Construction. We begin with some definitions. Let W be a *finite* weight-sequence. By abuse of notation, the discounted-sum of finite-sequence W with discount-factor d is defined as $DS(W, d) = DS(W \cdot 0^\omega, d)$. The *recoverable-gap* of a finite weight-sequences W with discount factor d , denoted $\text{gap}(W, d)$, is its normalized discounted-sum: If $W = \varepsilon$ (the empty sequence), $\text{gap}(\varepsilon, d) = 0$, and $\text{gap}(W, d) = d^{|W|-1} \cdot DS(W, d)$ otherwise [15]. Observe that the recoverable-gap has an inductive definition i.e. $\text{gap}(\varepsilon, d) = 0$, where ε is the empty weight-sequence, and $\text{gap}(W \cdot v, d) = d \cdot \text{gap}(W, d) + v$, where $v \in \{-\mu, \dots, \mu\}$.

This observation influences a sketch for $\text{bad}(\mu, d, \leq)$. Suppose all possible values for recoverable-gap of weight sequences forms the set of states. Then, the transition relation of the DFA can mimic the inductive definition of recoverable gap i.e. there is a transition from state s to t on alphabet $v \in \{-\mu, \dots, \mu\}$ iff $t = d \cdot s + v$, where s and v are recoverable-gap values of weight-sequences. There is one caveat here: There are infinitely many possibilities for the values of recoverable gap. We need to limit the recoverable gap values to finitely many values of interest. The core aspect of this construction is to identify these values.

First, we obtain a lower bound on recoverable gap for bad-prefixes of $\mathcal{A}_{\leq}^{\mu,d}$:

Lemma 1. *Let μ and $d > 1$ be the bound and discount-factor, resp. Let $T = \frac{\mu}{d-1}$ be the threshold value. Let W be a non-empty, bounded, finite weight-sequence. Weight sequence W is a bad-prefix of $\mathcal{A}_{\leq}^{\mu,d}$ iff $\text{gap}(W, d) > T$.*

Proof. Let a finite weight-sequence W be a bad-prefix of $\mathcal{A}_{\leq}^{\mu,d}$. Then, $DS(W \cdot Y, d) > 0$ for all infinite and bounded weight-sequences Y . Since $DS(W \cdot Y, d) = DS(W, d) + \frac{1}{d^{|W|}} \cdot DS(Y, d)$, we get $\inf(DS(W, d)) + \frac{1}{d^{|W|}} \cdot DS(Y, d) > 0 \implies DS(W, d) + \frac{1}{d^{|W|}} \cdot \inf(DS(Y, d)) > 0$ as W is a fixed sequence. Hence $DS(W, d) + \frac{-T}{d^{|W|-1}} > 0 \implies \text{gap}(W, d) - T > 0$. Conversely, for all infinite, bounded, weight-sequence Y , $DS(W \cdot Y, d) \cdot d^{|W|-1} = \text{gap}(W, d) + \frac{1}{d} \cdot DS(Y, d)$. Since $\text{gap}(W, d) > T$, $\inf(DS(Y, d)) = -T \cdot d$, we get $DS(W \cdot Y, d) > 0$. \square

Since all finite and bounded extensions of bad-prefixes are also bad-prefixes, Lemma 1 implies that if the recoverable-gap of a finite sequence is strictly lower than threshold T , then recoverable gap of all of its extensions also exceed T . Since recoverable gap exceeding threshold T is the precise condition for bad-prefixes, all states with recoverable gap exceeding T can be merged into a single state. Note, this state forms an accepting sink in $\text{bad}(\mu, d, \leq)$.

Next, we attempt to merge very low recoverable gap value into a single state. For this purpose, we define *very-good prefixes* for $\mathcal{A}_{\leq}^{\mu,d}$: A finite and bounded weight-sequence W is a *very good* prefix for language of $\mathcal{A}_{\leq}^{\mu,d}$ if for all infinite, bounded extensions of W by Y , $DS(W \cdot Y, d) \leq 0$. A proof similar to Lemma 1 proves an upper bound for the recoverable gap of very-good prefixes of $\mathcal{A}_{\leq}^{\mu,d}$:

Lemma 2. *Let μ and $d > 1$ be the bound and discount-factor, resp. Let $T = \frac{\mu}{d-1}$ be the threshold value. Let W be a non-empty, bounded, finite weight-sequence. Weight-sequence W is a very-good prefix of $\mathcal{A}_{\leq}^{\mu,d}$ iff $\text{gap}(W, d) \leq -T$.*

Clearly, finite extensions of very-good prefixes are also very-good prefixes. Further, $\text{bad}(\mu, d, \leq)$ must not accept very-good prefixes. Thus, by reasoning as earlier we get that all recoverable gap values that are less than or equal to $-T$ can be merged into one non-accepting sink state in $\text{bad}(\mu, d, \leq)$.

Finally, for an integer discount-factor the recoverable gap is an integer. Let $\lfloor x \rfloor$ denote the floor of $x \in \mathbb{R}$ e.g. $\lfloor 2.3 \rfloor = 2$, $\lfloor -2 \rfloor = -2$, $\lfloor -2.3 \rfloor = -3$. Then,

Corollary 2. *Let μ be the bound and $d > 1$ an integer discount-factor. Let $T = \frac{\mu}{d-1}$ be the threshold. Let W be a non-empty, bounded, finite weight-sequence.*

- W is a bad prefix of $\mathcal{A}_{\leq}^{\mu,d}$ iff $\text{gap}(W, d) > \lfloor T \rfloor$
- W is a very-good prefix of $\mathcal{A}_{\leq}^{\mu,d}$ iff $\text{gap}(W, d) \leq \lfloor -T \rfloor$

So, the recoverable gap value is either one of $\{\lfloor -T \rfloor + 1, \dots, \lfloor T \rfloor\}$, or less than or equal to $\lfloor -T \rfloor$, or greater than $\lfloor T \rfloor$. This curbs the state-space to $\mathcal{O}(\mu)$ -many values of interest, as $T = \frac{\mu}{d-1} < \frac{\mu \cdot d}{d-1}$ and $1 < \frac{d}{d-1} \leq 2$. Lastly, since $\text{gap}(\varepsilon, d) = 0$, state 0 must be the initial state.

Construction of $\text{bad}(\mu, d, \leq)$. Let μ be the upper bound, and $d > 1$ be the integer discount-factor. Let $T = \frac{\mu}{d-1}$ be the threshold value. The finite-state automata $\text{bad}(\mu, d, \leq) = (S, s_I, \Sigma, \delta, \mathcal{F})$ is defined as follows:

- States $S = \{\lfloor -T \rfloor + 1, \dots, \lfloor T \rfloor\} \cup \{\text{bad}, \text{veryGood}\}$
- Initial state $s_I = 0$, Accepting states $\mathcal{F} = \{\text{bad}\}$
- Alphabet $\Sigma = \{-\mu, -\mu + 1, \dots, \mu - 1, \mu\}$
- Transition function $\delta \subseteq S \times \Sigma \rightarrow S$ where $(s, a, t) \in \delta$ then:
 1. If $s \in \{\text{bad}, \text{veryGood}\}$, then $t = s$ for all $a \in \Sigma$
 2. If $s \in \{\lfloor -T \rfloor + 1, \dots, \lfloor T \rfloor\}$, and $a \in \Sigma$
 - (a) If $\lfloor -T \rfloor < d \cdot s + a \leq \lfloor T \rfloor$, then $t = d \cdot s + a$
 - (b) If $d \cdot s + a > \lfloor T \rfloor$, then $t = \text{bad}$
 - (c) If $d \cdot s + a \leq \lfloor -T \rfloor$, then $t = \text{veryGood}$

Theorem 2. *Let μ be the upper bound, $d > 1$ be the integer discount-factor. $\text{bad}(\mu, d, \leq)$ accepts finite, bounded, weight-sequence iff it is a bad-prefix of $\mathcal{A}_{\leq}^{\mu,d}$.*

Proof (Proof sketch). First note that the transition relation is deterministic and complete. Therefore, every word has a unique run in $\text{bad}(\mu, d, \leq)$. Let `last` be

the last state in the run of finite, bounded, weight-sequence W in the DFA. Use induction on the length of W to prove the following:

- $\text{last} \in \{-\lceil T \rceil + 1, \dots, \lfloor T \rfloor\}$ iff $\text{gap}(W, d) = \text{last}$
- $\text{last} = \text{bad}$ iff $\text{gap}(W, d) > \lfloor T \rfloor$
- $\text{last} = \text{veryGood}$ iff $\text{gap}(W, d) \leq -\lceil T \rceil$

Therefore, a finite, bounded weight-sequence is accepted iff its recoverable gap is greater than $\lfloor T \rfloor$. In other words, iff it is a bad-prefix of $\mathcal{A}_{\leq}^{\mu, d}$. \square

$\mathcal{A}_{\leq}^{\mu, d}$ is obtained from $\text{bad}(\mu, d, \leq)$ by applying co-Büchi acceptance condition.

Theorem 3. *Let μ be the upper bound, and $d > 1$ be the integer discount-factor. DS-comparator for all inequalities and equality are either deterministic safety or deterministic co-safety automata with $\mathcal{O}(\mu)$ states.*

As a matter of fact, the most compact non-deterministic DS-comparator constructions with parameters μ , d and R also contain $\mathcal{O}(\mu)$ states [11].

3.3 Quantitative Inclusion with Safety/Co-safety Comparators

This section investigates quantitative language inclusion with regular safety/co-safety comparators. Unlike quantitative inclusion with regular comparators, quantitative inclusion with regular safety/co-safety comparators is able to circumvent Büchi complementation with intermediate subset-construction steps. As a result, complexity of quantitative inclusion with regular safety/co-safety comparator is lower than the same with regular comparators [12] (Theorem 4). Finally, since DS-comparators are regular safety/co-safety comparators, the algorithm for quantitative inclusion with regular safety/co-safety comparators applies to DS-inclusion yielding a lower complexity algorithm for DS-inclusion (Corollary 5).

Key Ideas A run of word w in a weighted-automaton is *maximal* if its weight is the supremum weight of all runs of w in the weighted-automaton. A run ρ_P of w in P is a *counterexample* for $P \subseteq Q$ (or $P \subset Q$) iff there exists a maximal run sup_Q of w in Q such that $\text{wt}(\rho_P) > \text{wt}(\text{sup}_Q)$ (or $\text{wt}(\rho_P) \geq \text{wt}(\text{sup}_Q)$). Consequently, $P \subseteq Q$ (or $P \subset Q$) iff there are no counterexample runs in P . Therefore, the roadmap to solve quantitative inclusion for regular safety/co-safety comparators is as follows:

1. Use regular safety/co-safety comparators to construct the *maximal automaton* of Q i.e. an automaton that accepts all maximal runs of Q (Corollary 3).
2. Use the regular safety/co-safety comparator and the maximal automaton to construct a *counterexample automaton* that accepts all counterexample runs of the inclusion problem $P \subseteq Q$ (or $P \subset Q$) (Lemma 5).

3. Solve quantitative inclusion for safety/co-safety comparator by checking for emptiness of the counterexample (Theorem 4).

Finally, since DS-comparators are regular safety/co-safety automaton (Corollary 1), apply Theorem 4 to obtain an algorithm for DS-inclusion that uses regular safety/co-safety comparators (Corollary 5).

Let W be a weighted automaton. Then the *annotated automaton* of W , denoted by \hat{W} , is the Büchi automaton obtained by transforming transition $s \xrightarrow{a} t$ with weight v in W to transition $s \xrightarrow{a,v} t$ in \hat{W} . Observe that \hat{W} is a safety automaton since all its states are accepting. A run on word w with weight sequence wt in W corresponds to an *annotated word* (w, wt) in \hat{W} , and vice-versa.

Maximal Automaton. This section covers the construction of the *maximal automaton* from a weighted automaton. Let W and \hat{W} be a weighted automaton and its annotated automaton, respectively. We call an annotated word (w, wt_1) in \hat{W} *maximal* if for all other words of the form (w, wt_2) in \hat{W} , $wt(wt_1) \geq wt(wt_2)$. Clearly, (w, wt_1) is a maximal word in \hat{W} iff word w has a run with weight sequence wt_1 in W that is maximal. We define *maximal automaton* of weighted automaton W , denoted $\text{Maximal}(W)$, to be the automaton that accepts all maximal words of its annotated automata \hat{W} .

We show that when the comparator is regular safety/co-safety, the construction of the maximal automata incurs a $2^{\mathcal{O}(n)}$ blow-up. This section exposes the construction for maximal automaton when comparator for non-strict inequality is regular safety. The other case when the comparator for strict inequality is regular co-safety has been deferred to the appendix.

Lemma 3. *Let W be a weighted automaton with regular safety comparator for non-strict inequality. Then the language of $\text{Maximal}(W)$ is a safety language.*

Proof (Proof sketch). An annotated word (w, wt_1) is not maximal in \hat{W} for one of the following two reasons: Either (w, wt_1) is not a word in \hat{W} , or there exists another word (w, wt_2) in \hat{W} s.t. $wt(wt_1) < wt(wt_2)$ (equivalently (wt_1, wt_2) is not in the comparator non-strict inequality). Both \hat{W} and comparator for non-strict inequality are safety languages, so the language of maximal words must also be a safety language. \square

We now proceed to construct the safety automata for $\text{Maximal}(W)$

Intuition. The intuition behind the construction of maximal automaton follows directly from the definition of maximal words. Let \hat{W} be the annotated automaton for weighted automaton W . Let $\hat{\Sigma}$ denote the alphabet of \hat{W} . Then an annotated word $(w, wt_1) \in \hat{\Sigma}^\omega$ is a word in $\text{Maximal}(W)$ if (a) $(w, wt_1) \in \hat{W}$, and (b) For all words $(w, wt_2) \in \hat{W}$, $wt(wt_1) \geq wt(wt_2)$.

The challenge here is to construct an automaton for condition (b). Intuitively, this automaton simulates the following action: As the automaton reads word (w, wt_1) , it must spawn all words of the form (w, wt_2) in \hat{W} , while also ensuring that $wt(wt_1) \geq wt(wt_2)$ holds for every word (w, wt_2) in \hat{W} . Since \hat{W} is a safety

automaton, for a word $(w, wt_1) \in \hat{\Sigma}^\omega$, all words of the form $(w, wt_2) \in \hat{W}$ can be traced by subset-construction. Similarly since the comparator C for non-strict inequality (\geq) is a safety automaton, all words of the form $(wt_1, wt_2) \in C$ can be traced by subset-construction as well. The construction needs to carefully align the word (w, wt_1) with the all possible $(w, wt_2) \in \hat{W}$ and $(wt_1, wt_2) \in C$.

Construction of Maximal(W). Let W be a weighted automaton, with annotated automaton \hat{W} and C denote its regular safety comparator for non-strict inequality. Let S_W denote the set of states of W (and \hat{W}) and S_C denote the set of states of C . We define $\text{Maximal}(W) = (S, s_I, \hat{\Sigma}, \delta, \mathcal{F})$ as follows:

- Set of states S consists of tuples of the form (s, X) , where $s \in S_W$, and $X = \{(t, c) | t \in S_W, c \in S_C\}$
- $\hat{\Sigma}$ is the alphabet of \hat{W}
- Initial state $s_I = (s_w, \{(s_w, s_c)\})$, where s_w and s_c are initial states in \hat{W} and C , respectively.
- Let states $(s, X), (s, X') \in S$ such that $X = \{(t_1, c_1), \dots, (t_n, c_n)\}$ and $X' = \{(t'_1, c'_1), \dots, (t'_m, c'_m)\}$. Then $(s, X) \xrightarrow{(a,v)} (s', X') \in \delta$ iff
 1. $s \xrightarrow{(a,v)} s'$ is a transition in \hat{W} , and
 2. $(t'_j, c'_j) \in X'$ if there exists $(t_i, c_i) \in X$, and a weight v' such that $t_i \xrightarrow{a,v'} t'_j$ and $c_i \xrightarrow{v,v'} c'_j$ are transitions in \hat{W} and C , respectively.
- $(s, \{(t_1, c_1), \dots, (t_n, c_n)\}) \in \mathcal{F}$ iff s and all t_i are accepting in \hat{W} , and all c_i is accepting in C .

Lemma 4. *Let W be a weighted automaton with regular safety comparator C for non-strict inequality. Then the size of $\text{Maximal}(W)$ is $|W| \cdot 2^{\mathcal{O}(|W| \cdot |C|)}$.*

Proof (Proof sketch). A state $(s, \{(t_1, c_1), \dots, (t_n, c_n)\})$ is non-accepting in the automata if one of s, t_i or c_j is non-accepting in underlying automata \hat{W} and the comparator. Since \hat{W} and the comparator automata are safety, all outgoing transitions from a non-accepting state go to non-accepting state in the underlying automata. Therefore, all outgoing transitions from a non-accepting state in $\text{Maximal}(W)$ go to non-accepting state in $\text{Maximal}(W)$. Therefore, $\text{Maximal}(W)$ is a safety automaton. To see correctness of the transition relation, one must prove that transitions of type (1.) satisfy condition (a), while transitions of type (2.) satisfy condition (b). $\text{Maximal}(W)$ forms the conjunction of (a) and (b), hence accepts the language of maximal words of W .

A similar construction proves that the maximal automata of weighted automata W with regular safety comparator C for strict inequality contains $|W| \cdot 2^{\mathcal{O}(|W| \cdot |C|)}$ states. In this case, however, the maximal automaton may not be a safety automaton. Therefore, Lemma 4 generalizes to:

Corollary 3. *Let W be a weighted automaton with regular safety/co-safety comparator C . Then $\text{Maximal}(W)$ is a Büchi automaton of size $|W| \cdot 2^{\mathcal{O}(|W| \cdot |C|)}$.*

Counterexample Automaton. This section covers the construction of the counterexample automaton. Given weighted-automata P and Q , an annotated word (w, wt_P) in annotated automata \hat{P} is a *counterexample word* of $P \subseteq Q$ (or $P \subset Q$) if there exists (w, wt_Q) in $\text{Maximal}(Q)$ s.t. $wt(wt_P) > wt(wt_Q)$ (or $wt(wt_P) \geq wt(wt_Q)$). Clearly, annotated word (w, wt_P) is a counterexample word iff there exists a counterexample run of w with weight-sequence wt_P in P .

For this section, we abbreviate strict and non-strict to `strct` and `nstrct`, respectively. For $\text{inc} \in \{\text{strct}, \text{nstrct}\}$, the *counterexample automaton* for inc -quantitative inclusion, denoted by $\text{Counterexample}(\text{inc})$, is the automaton that contains all counterexample words of the problem instance. We construct the counterexample automaton as follows:

Lemma 5. *Let P, Q be weighted-automata with regular safety/co-safety comparators. For $\text{inc} \in \{\text{strct}, \text{nstrct}\}$, $\text{Counterexample}(\text{inc})$ is a Büchi automaton.*

Proof. We construct Büchi automaton $\text{Counterexample}(\text{inc})$ for $\text{inc} \in \{\text{strct}, \text{nstrct}\}$ that contains the counterexample words of inc -quantitative inclusion. Since the comparator are regular safety/co-safety, $\text{Maximal}(Q)$ is a Büchi automaton (Corollary 3). Construct the product $\hat{P} \times \text{Maximal}(Q)$ such that transition $(p_1, q_1) \xrightarrow{a, v_1, v_2} (p_1, q_2)$ is in the product iff $p_1 \xrightarrow{a, v_1} p_1$ and $q_1 \xrightarrow{a, v_2} q_2$ are transitions in \hat{P} and $\text{Maximal}(Q)$, respectively. A state (p, q) is accepting if both p and q are accepting in \hat{P} and $\text{Maximal}(Q)$. One can show that the product accepts (w, wt_P, wt_Q) iff (w, wt_P) and (w, wt_Q) are words in \hat{P} and $\text{Maximal}(Q)$, respectively.

If $\text{inc} = \text{strct}$, intersect $\hat{P} \times \text{Maximal}(Q)$ with comparator for \geq . If $\text{inc} = \text{nstrct}$, intersect $\hat{P} \times \text{Maximal}(Q)$ with comparator for $>$. Since the comparator is a safety or co-safety automaton, the intersection is taken without the cyclic counter. Therefore, $(s_1, t_1) \xrightarrow{a, v_1, v_2} (s_2, t_2)$ is a transition in the intersection iff $s_1 \xrightarrow{a, v_1, v_2} s_2$ and $t_1 \xrightarrow{v_1, v_2} t_2$ are transitions in the product and the appropriate comparator, respectively. State (s, t) is accepting if both s and t are accepting. The intersection will accept (w, wt_P, wt_Q) iff (w, wt_P) is a counterexample of inc -quantitative inclusion. $\text{Counterexample}(\text{inc})$ is obtained by projecting out the intersection as follows: Transition $m \xrightarrow{a, v_1, v_2} n$ is transformed to $m \xrightarrow{a, v_1} n$. \square

Quantitative Inclusion and DS-inclusion. In this section, we give the final algorithm for quantitative inclusion with regular safety/co-safety comparators. Since DS-comparators are regular safety/co-safety comparators, this gives us an algorithm for DS-inclusion with improved complexity than previous results.

Theorem 4. *Let P, Q be weighted-automata with regular safety/co-safety comparators. Let C_{\leq} and $C_{<}$ be the comparators for \leq and $<$, respectively. Then*

- Strict quantitative inclusion $P \subset Q$ is reduced to emptiness checking of a Büchi automaton of size $|P||C_{\leq}||Q| \cdot 2^{\mathcal{O}(|Q| \cdot |C_{\leq}|)}$.
- Non-strict quantitative inclusion $P \subseteq Q$ is reduced to emptiness checking of a Büchi automaton of size $|P||C_{<}||Q| \cdot 2^{\mathcal{O}(|Q| \cdot |C_{<}|)}$.

Proof. Strict and non-strict are abbreviated to `strct` and `nstrct`, respectively. For $\text{inc} \in \{\text{strct}, \text{nstrct}\}$, inc -quantitative inclusion holds iff $\text{Counterexample}(\text{inc})$ is empty. Size of $\text{Counterexample}(\text{inc})$ is the product of size of P , $\text{Maximal}(Q)$ (Corollary 3), and the appropriate comparator as described in Lemma 5. \square

In contrast, quantitative inclusion with regular comparators reduces to emptiness of a Büchi automaton with $|P| \cdot 2^{\mathcal{O}(|P||Q||C| \cdot \log(|P||Q||C|))}$ states [12]. The $2^{\mathcal{O}(n \log n)}$ blow-up is unavoidable due to Büchi complementation. Hence, quantitative inclusion with regular safety/co-safety has lower worst-case complexity.

Lastly, we use the results of developed in previous sections to solve DS-inclusion. Since DS-comparators are regular safety/co-safety (Corollary 1), an immediate consequence of Theorem 4 is an improvement in the worst-case complexity of DS-inclusion in comparison to prior results with regular DS-comparators. Furthermore, since the regular safety/co-safety DS-comparators are of the same size for all inequalities (Theorem 3), we get:

Corollary 4. *Let P, Q be weighted-automata, and C be a regular safety/co-safety DS-comparator with integer discount-factor $d > 1$. Strict DS-inclusion reduces to emptiness checking of a safety automaton of size $|P||C||Q| \cdot 2^{\mathcal{O}(|Q| \cdot |C|)}$.*

Proof (Proof sketch). When comparator for non-strict inequality is safety-automaton, as it is for DS-comparator, the maximal automaton is a safety automaton (Lemma 3). One can then show that the counterexample automata is also a safety automaton.

A similar argument proves *non-strict DS-inclusion* reduces to emptiness of a *weak-Büchi automaton* [27] of size $|P||C||Q| \cdot 2^{\mathcal{O}(|Q| \cdot |C|)}$ (see Appendix).

Corollary 5 ([DS-inclusion with safety/co-safety comparator]). *Let P, Q be weighted-automata, and C be a regular (co)-safety DS-comparator with integer discount-factor $d > 1$. The complexity of DS-inclusion is $|P||C||Q| \cdot 2^{\mathcal{O}(|Q| \cdot |C|)}$.*

4 Implementation and Experimental Evaluation

The goal of the empirical analysis is to examine performance of DS-inclusion with integer discount-factor with safety/co-safety comparators against existing tools to investigate the practical merit of our algorithm. We compare against (a) Regular-comparator based tool QuIP, and (b) DS-determinization and linear-programming tool DetLP.

QuIP is written in C++, and invokes state-of-the-art Büchi language inclusion-solver RABIT [2]. We enable the `-fast` flag in RABIT, and tune its Java-threads with `Xss`, `Xms`, `Xmx` set to 1GB, 1GB and 8GB, respectively. DetLP is also written in C++, and uses linear programming solver GLPSOL provided by GLPK (GNU Linear Prog. Kit) [1]. We compare these tools along two axes: runtime and number of benchmarks solved.

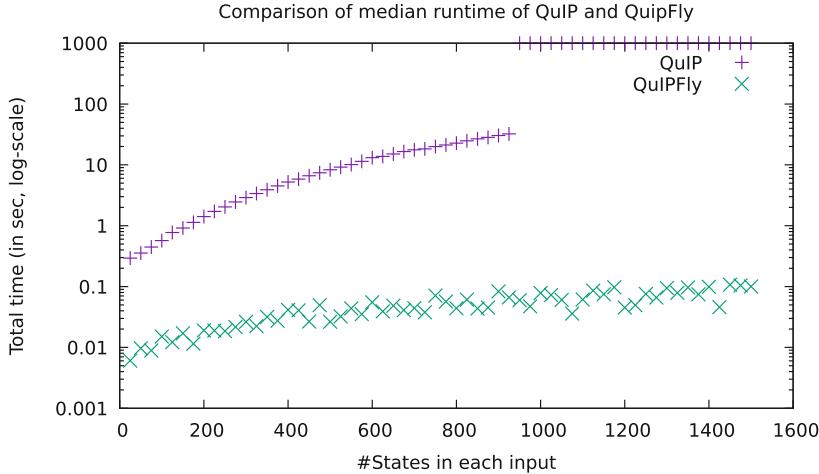


Fig. 1. $s_P = s_Q$ on x -axis, $wt = 4$, $\delta = 3$, $d = 3$, $P \subset Q$

Implementation Details. The algorithm for strict-DS-inclusion with integer discount factor $d > 1$ proposed in Corollary 4 and non-strict DS-inclusion checks for emptiness of the counterexample automata. A naive algorithm will construct the counterexample automata fully, and then check if they are empty by ensuring the absence of an *accepting lasso*.

We implement a more efficient algorithm. In our implementation, we make use of the fact that the constructions for DS-inclusion use subset-construction intermediate steps. This facilitates an *on-the-fly procedure* since successor states of state in the counterexample automata can be determined directly from input weighted automata and the comparator automata. The algorithm terminates as soon as an accepting lasso is detected. When an accepting lasso is absent, the algorithm traverses all states and edges of the counterexample automata.

We implement the optimized on-the-fly algorithm in a prototype QuIPFly. QuIPFly is written in Python 2.7.12. QuIPFly employs basic implementation-level optimizations to avoid excessive re-computation.

Design and Setup for Experiments. Due to lack of standardized benchmarks for weighted automata, we follow a standard approach to performance evaluation of automata-theoretic tools [3, 30, 38] by experimenting with *randomly generated* benchmarks, using random benchmark generation procedure described in [11].

The parameters for each experiment are number of states s_P and s_Q of weighted automata, transition density δ , maximum weight wt , integer discount-factor d , and $inc \in \{\text{strct}, \text{nstrct}\}$. In each experiment, weighted automata P and Q are randomly generated, and runtime of inc -DS-inclusion for all three tools is reported with a timeout of 900 s. We run the experiment for each parameter tuple 50 times. All experiments are run on a single node of a high-performance cluster consisting of two quad-core Intel-Xeon processor running at 2.83 GHz,

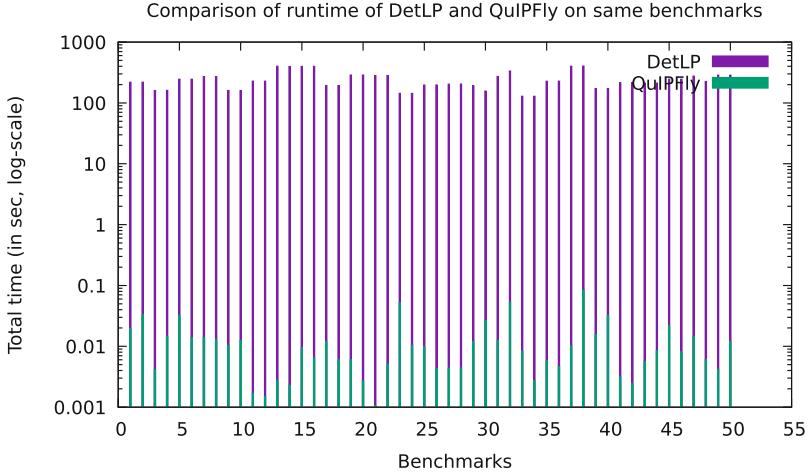


Fig. 2. $s_P = s_Q = 75$, $wt = 4$, $\delta = 3$, $d = 3$, $P \subset Q$

with 8 GB of memory per node. We experiment with $s_P = s_Q$ ranging from 0–1500 in increments of 25, $\delta \in \{3, 3.5, 4\}$, $d = 3$, and $wt \in \{d^1 + 1, d^3 - 1, d^4 - 1\}$.

Observations and Inferences.¹ For clarity of exposition, we present the observations for only one parameter-tuple. Trends and observations for other parameters were similar.

QuIPFly Outperforms. QuIP by at least an order of magnitude in runtime. Figure 1 plots the median runtime of all 50 experiments for the given parameter-values for QuIP and QuIPFly. More importantly, QuIPFly solves all of our benchmarks within a fraction of the timeout, whereas QuIP struggled to solve at least 50% of the benchmarks with larger inputs (beyond $s_P = s_Q = 1000$). Primary cause of failure is memory overflow inside RABIT. We conclude that regular safety/co-safety comparators outperform their regular counterpart, giving credit to the simpler subset-constructions vs. Büchi complementation.

QuIPFly Outperforms. *DetLP* comprehensively in runtime and in number of benchmarks solved. We were unable to plot *DetLP* in Fig. 1 since it solved fewer than 50% benchmarks even with small input instances. Figure 2 compares the runtime of both tools on the same set of 50 benchmarks for a representative parameter-tuple on which all 50 benchmarks were solved. The plot shows that QuIPFly beats *DetLP* by 2–4 orders of magnitude on all benchmarks.

Overall Verdict. Overall, QuIPFly outperforms QuIP and *DetLP* by a significant margin along both axes, runtime and number of benchmarks solved. This analysis gives unanimous evidence in favor of our safety/co-safety approach to solving DS-inclusion.

¹ Figures are best viewed online and in color.

5 Concluding Remarks

The goal of this paper was to build scalable algorithms for DS-inclusion. To this end, this paper furthers the understanding of language-theoretic properties of discounted-sum aggregate function by demonstrating that DS-comparison languages form safety and co-safety languages, and utilizes these properties to obtain a decision procedure for DS-inclusion that offers both tighter theoretical complexity and improved scalability. All in all, the key insights of this work are:

1. Pure automata-theoretic techniques of DS-comparator are better for DS-inclusion;
2. In-depth language-theoretic analysis improve both theoretical complexity and practical scalability of DS-inclusion;
3. DS-comparators are compact deterministic safety or co-safety automata.

To the best of our knowledge, this is the first work that applies language-theoretic properties such as safety/co-safety in the context of quantitative reasoning.

More broadly, this paper demonstrates that the close integration of language-theoretic and quantitative properties can render novel algorithms for quantitative reasoning that can benefit from advances in qualitative reasoning.

Acknowledgements. We thank anonymous reviewers for their comments. We thank D. Fried, L. M. Tabajara, and A. Verma for their valuable inputs on initial drafts of the paper. This work was partially supported by NSF Grant No. CCF-1704883.

References

1. GLPK. <https://www.gnu.org/software/glpk/>
2. Rabbit-Reduce. <http://www.languageinclusion.org/>
3. Abdulla, P.A., et al.: Simulation subsumption in ramsey-based büchi automata universality and inclusion testing. In: Proceedings of CAV, pp. 132–147. Springer (2010)
4. Abdulla, P.A., et al.: Advanced ramsey-based büchi automata inclusion testing. In: Proceedings of CONCUR, vol. 11, pp. 187–202. Springer (2011)
5. Alpern, B., Schneider, F.B.: Recognizing safety and liveness. Distrib. Comput. **2**(3), 117–126 (1987)
6. Alur, R., Mamouras, K.: An introduction to the streamqre language. Dependable Softw. Syst. Eng. **50**, 1 (2017)
7. Aminof, B., Kupferman, O., Lampert, R.: Reasoning about online algorithms with weighted automata. Trans. Algorithms **6**(2), 28 (2010)
8. Andersen, G., Conitzer, V.: Fast equilibrium computation for infinitely repeated games. In: Proceedings of AAAI, pp. 53–59 (2013)
9. Andersson, D.: An improved algorithm for discounted payoff games. In: ESSLLI Student Session, pp. 91–98 (2006)
10. Baier, C.: Probabilistic model checking. In: Dependable Software Systems Engineering, pp. 1–23 (2016)
11. Bansal, S., Chaudhuri, S., Vardi, M.Y.: Automata vs linear-programming discounted-sum inclusion. In: Proceedings of International Conference on Computer-Aided Verification (CAV) (2018)

12. Bansal, S., Chaudhuri, S., Vardi, M.Y. : Comparator automata in quantitative verification. In: Proceedings of International Conference on Foundations of Software Science and Computation Structures (FoSSaCS) (2018)
13. Bansal, S., Chaudhuri, S., Vardi, M.Y.: Comparator automata in quantitative verification (full version). CoRR, abs/1812.06569 (2018)
14. Bloem, R., Chatterjee, K., Henzinger, T.A., Jobstmann, B.: Better quality in synthesis through quantitative objectives. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 140–156. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_14
15. Boker, U., Henzinger, T.A.: Exact and approximate determinization of discounted-sum automata. LMCS **10**(1), 1–13 (2014)
16. Chakrabarti, A., Chatterjee, K., Henzinger, T.A., Kupferman, O., Majumdar, R.: Verifying quantitative properties using bound functions. In: Borrione, D., Paul, W. (eds.) CHARME 2005. LNCS, vol. 3725, pp. 50–64. Springer, Heidelberg (2005). https://doi.org/10.1007/11560548_7
17. Chatterjee, K., Doyen, L., Henzinger, T.A.: Quantitative languages. Trans. Computat. Logic **11**(4), 23 (2010)
18. Chaudhuri, S., Sankaranarayanan, S., Vardi, M.Y.: Regular real analysis. In: Proceedings of LICS, pp. 509–518 (2013)
19. de Alfaro, L., Henzinger, T.A., Majumdar, R.: Discounting the future in systems theory. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 1022–1037. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-45061-0_79
20. Doyen, L., Raskin, J.-F.: Antichain algorithms for finite automata. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 2–22. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12002-2_2
21. Droste, M., Kuich, W., Vogler, H.: Handbook of Weighted Automata. Springer, Berlin (2009)
22. D’Antoni, L., Samanta, R., Singh, R.: Qlose: program repair with quantitative objectives. In: Proceedings of CAV, pp. 383–401. Springer (2016)
23. Filiot, E., Gentilini, R., Raskin, J.-F.: Quantitative languages defined by functional automata. In: Koutny, M., Ulidowski, I. (eds.) CONCUR 2012. LNCS, vol. 7454, pp. 132–146. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32940-1_11
24. He, K., Lahijanian, M., Kavraki, L.E., Vardi, M.Y.: Reactive synthesis for finite tasks under resource constraints. In: 2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp. 5326–5332. IEEE (2017)
25. Hu, Q., DAntoni, L.: Syntax-guided synthesis with quantitative syntactic objectives. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 386–403. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_21
26. Kupferman, O., Vardi, M.Y.: Model checking of safety properties. In: Halbwachs, N., Peled, D. (eds.) CAV 1999. LNCS, vol. 1633, pp. 172–183. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48683-6_17
27. Kupferman, O., Vardi, M.Y.: Weak alternating automata are not that weak. Trans. Computat. Logic **2**(3), 408–429 (2001)
28. Kwiatkowska, M.: Quantitative verification: models, techniques and tools. In: Proceedings 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE), pp. 449–458. ACM Press, September 2007

29. Lahijanian, M., Almagor, S., Fried, D., Kavraki, L.E., Vardi, M.Y.: This time the robot settles for a cost: a quantitative approach to temporal logic planning with partial satisfaction. In: AAAI, pp. 3664–3671 (2015)
30. Mayr, R., Clemente, L.: Advanced automata minimization. ACM SIGPLAN Not. **48**(1), 63–74 (2013)
31. Mohri, M.: Weighted automata algorithms. In: Droste, M., Kuich, W., Vogler, H. (eds.) Handbook of Weighted Automata. Monographs in Theoretical Computer Science. An EATCS Series. Springer, Berlin (2009). https://doi.org/10.1007/978-3-642-01492-5_6
32. Mohri, M., Pereira, F., Riley, M.: Weighted finite-state transducers in speech recognition. Comput. Speech Lang. **16**(1), 69–88 (2002)
33. Osborne, M.J., Rubinstein, A.: A Course in Game Theory. MIT press, Cambridge (1994)
34. Puterman, M.L.: Markov decision processes. Handbooks Oper. Res. Manag. Sci. **2**, 331–434 (1990)
35. Rudin, W.: Principles of Mathematical Analysis, vol. 3. McGraw-Hill, New York (1964)
36. Safra, S.: On the complexity of ω -automata. In: Proceedings of FOCS, pp. 319–327. IEEE (1988)
37. Sutton, R.S., Barto, A.G.: Introduction to Reinforcement Learning, vol. 135. MIT press, Cambridge (1998)
38. Tabakov, D., Vardi, M.Y.: Experimental evaluation of classical automata constructions. In: Sutcliffe, G., Voronkov, A. (eds.) LPAR 2005. LNCS (LNAI), vol. 3835, pp. 396–411. Springer, Heidelberg (2005). https://doi.org/10.1007/11591191_28
39. Thomas, W., Wilke, T., et al.: Automata, Logics, and Infinite Games: A Guide to Current Research, vol. 2500. Springer Science & Business Media, Berlin (2002)
40. Vardi, M.Y.: The Büchi complementation saga. In: Annual Symposium on Theoretical Aspects of Computer Science, pp. 12–22. Springer (2007)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Clock Bound Repair for Timed Systems

Martin Kölbl^{1(✉)}, Stefan Leue^{1(✉)}, and Thomas Wies^{2(✉)}

¹ University of Konstanz, Konstanz, Germany

{Martin.Koelbl,Stefan.Leue}@uni-konstanz.de

² New York University, New York, NY, USA

wies@cs.nyu.edu

Abstract. We present algorithms and techniques for the repair of timed system models, given as networks of timed automata (NTA). The repair is based on an analysis of timed diagnostic traces (TDTs) that are computed by real-time model checking tools, such as UPPAAL, when they detect the violation of a timed safety property. We present an encoding of TDTs in linear real arithmetic and use the MaxSMT capabilities of the SMT solver Z3 to compute possible repairs to clock bound values that minimize the necessary changes to the automaton. We then present an admissibility criterion, called functional equivalence, that assesses whether a proposed repair is admissible in the overall context of the NTA. We have implemented a proof-of-concept tool called TARTAR for the repair and admissibility analysis. To illustrate the method, we have considered a number of case studies taken from the literature and automatically injected changes to clock bounds to generate faulty mutations. Our technique is able to compute a feasible repair for 91% of the faults detected by UPPAAL in the generated mutants.

Keywords: Timed automata · Automated repair · Admissibility of repair · TARTAR tool

1 Introduction

The analysis of system design models using model checking technology is an important step in the system design process. It enables the automated verification of system properties against given design models. The automated nature of model checking facilitates the integration of the verification step into the design process since it requires no further intervention of the designer once the model has been formulated and the property has been specified.

Often it is sufficient to abstract from real time aspects when checking system properties, in particular when the focus is on functional aspects of the system. However, when non-functional properties, such as response times or the timing of periodic behavior, play an important role, it is necessary to incorporate real time aspects into the models and the specification, as well as to use specialized real-time model checking tools, such as UPPAAL [6], Kronos [31] or opaal [11] during the verification step.

Next to the automatic nature of model checking, the ability to return counterexamples, in real-time model checking often referred to as timed diagnostic traces (TDT), is

a further practical benefit of the use of model checking technology. A TDT describes a timed sequence of steps that lead the design model from the initial state of the system into a state violating a real-time property. A TDT neither constitutes a causal explanation of the property violation, nor does it provide hints as to how to correct the model.

In this paper we describe an automated method that computes proposals for possible repairs of a network of timed automata (NTA) that avoid the violation of a timed safety property. Consider the TDT depicted as a time annotated sequence diagram [5] in Fig. 1. This scenario describes a simple message exchange where the process dbServer sends a message `req` to process `db` which, after some processing steps returns a message `ser` to `dbServer`. Assume a requirement on the system to be that the time from sending `req` to receiving `ser` is not to be more than 4 time units. Assume that the timing interval annotations on the sequence diagram represent the minimum and maximum time for the message transmission and processing steps that the NTA, from which the diagram has been derived, permits. It is then easy to see that it is possible to execute the system in such a way that this property is violated.

Various changes to the underlying NTA model, depicted in Fig. 2, may avoid this property violation. For instance, the maximum time it takes to transmit the `req` and `ser` messages can be constrained to be at most 1 time unit, respectively. Alternatively, it may be possible to avoid the property violation by reducing two of the three timings by 0.5 time units. In any case, proposing such changes to the model may either serve to correct clerical mistakes made during the editing of the model, or point to necessary changes in the dimensioning of its time resources, thus contributing to improved design space exploration.

The repair method described in this paper relies on an encoding of a TDT as a constraint system in linear real arithmetic. This encoding provides a symbolic abstract semantics for the TDT by constraining the sojourn time of the NTA in the locations visited along the trace. The constraint system is then augmented by auxiliary model variation variables which represent syntactic changes to the NTA model, for instance the variation of a location invariant condition or a transition guard. We assert that the thus modified constraint system implies the non-reachability of a violation. At the same time, we assert that the model variation variables have a value that implies that no change of the NTA model will occur, for instance by setting a clock bound variation variable to 0. This renders the resulting constraint system unsatisfiable.

In order to compute a repair, we derive a partial MaxSMT instance by turning the constraints that disable any repair into soft constraints. We solve this MaxSMT instance using the SMT solver Z3 [25]. If the MaxSMT instance admits a solution, the resulting model provides values of the model variation variables. These values indicate a repair

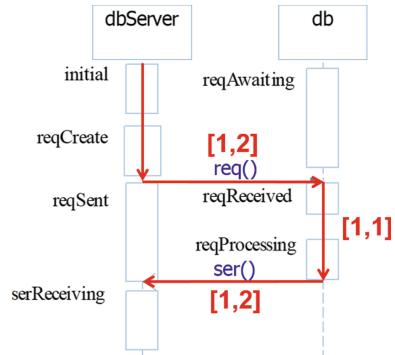


Fig. 1. TDT represented as a sequence diagram with timing annotations

of the NTA model which entails that along the sequence of locations represented by the TDT, the property violation will no longer be reachable.

In a next step it is necessary to check whether the computed repair is an admissible repair in the context of the full NTA. This is important since the repair was computed locally with respect to only a single given TDT. Thus, it is necessary to define a notion of admissibility that is reasonable and helpful in this setting. To this end, we propose the notion of *functional equivalence* which states that as a result of the computed repair, neither erstwhile existing functional behavior will be purged, nor will new functional behavior be added. Functional behavior in this sense is represented by languages accepted by the untimed automata of the unrepaired and the repaired NTAs. Functional equivalence is then defined as equivalence of the languages accepted by these automata. We propose a zone-based automaton construction for implementing the functional equivalence test that is efficient in practice.

We have implemented our proposed method in a proof-of-concept tool called TARTAR¹. Our evaluation of TARTAR is based on several non-trivial NTA models taken from the literature, including the frequently considered Pacemaker model [19]. For each model, we automatically generate mutants by injecting clock bound variations which we then model check using UPPAAL and repair using TARTAR. The evaluation shows that our technique is able to compute an admissible repair for 91% of the detected faults.

Related Work. There are relatively few results available on a formal treatment of TDTs. The zone based approach to real-time model checking, which relies on a constraint-based abstraction of the state space, is proposed in [14]. The use of constraint solving to perform reachability analysis for NTAs is described in [30]. This approach ultimately leads to the on-the-fly reachability analysis algorithm used in UPPAAL [7]. [12] defines the notion of a time-concrete UPPAAL counterexample. Work documented in [27] describes the computation of concrete delays for symbolic TDTs. The above cited approaches address neither fault analysis nor repair for TDTs. Our use of MaxSMT solvers for computing minimal repairs is inspired by the use MaxSAT solvers for fault localization in C programs, which was first explored in the BugAssist tool [20, 21]. Our approach also shares some similarities with syntax-guided synthesis [2, 28], which has also been deployed in the context of program repair [22]. One key difference is how we determine the admissibility of a repair in the overall system, which takes advantage of the semantic restrictions imposed by timed automata.

Structure of the Paper. We will introduce the automata and real-time concepts needed in our analysis in Sect. 2. In Sect. 3 we present the logical formalization of TDTs. The repair and admissibility analyses are presented in Sects. 4 and 5, respectively. We report on tool development, experimental evaluation and case studies in Sects. 6 and 7 concludes.

¹ TARTAR and links to all models used in this paper can be found at URL <https://github.com/sen-uni-kn/tartar>.

2 Preliminaries

The timed automaton model that we use in this paper is adapted from [7]. Given a set of *clocks* C , we denote by $\mathcal{B}(C)$ the set of all *clock constraints* over C , which are conjunctions of *atomic clock constraints* of the form $c \sim n$, where $c \in C$, $\sim \in \{<, \leq, =, \geq, >\}$ and $n \in \mathbb{N}$. A *timed automaton (TA)* T is a tuple $T = (L, l^0, C, \Sigma, \Theta, I)$ where L is a finite set of locations, $l^0 \in L$ is an initial location, C is a finite set of clocks, Σ is a set of action labels, $\Theta \subseteq_{fin} L \times \mathcal{B}(C) \times \Sigma \times 2^C \times L$ is a set of *actions*, and $I : L \rightarrow \mathcal{B}(C)$ denotes a labeling of locations with clock constraints, referred to as location invariants. For $\theta \in \Theta$ with $\theta = (l, g, a, r, l')$ we refer to g as the *guard* of θ and to r as its *clock resets*.

The operational semantics of T is given by a timed transition system consisting of states $s = (l, u)$ where l is a location and $u : C \rightarrow \mathbb{R}_+$ is a *clock valuation*. The initial state s_0 is (l_0, u_0) where u_0 maps all clocks to 0. For a clock constraint B we write $u \models B$ iff B evaluates to true in u . There are two types of transitions. An *action transition* models the execution of an action whose guard is satisfied. These transitions are instantaneous and reset the specified clocks. The passing of time in a location is modeled by *delay transitions*. Both types of transitions guarantee that location invariants are satisfied in the pre and post state. Formally, we have $(l, u) \xrightarrow{t} (l', u')$ iff

- (action transition) $t = (l, g, a, r, l') \in \Theta$, $u \models I(l) \wedge g$, $u' \models I(l')$ and for all clocks $c \in C$, $u'(c) = 0$ if $c \in r$ and $u'(c) = u(c)$ otherwise; or
- (delay transition) $t \in \mathbb{R}_+$, $u \models I(l)$, $u' \models I(l)$ and $u' = u + t$.

Definition 1. A symbolic timed trace (STT) of T is a sequence of actions $S = \theta_0, \dots, \theta_{n-1}$. A realization of S is a sequence of delay values $\delta_0, \dots, \delta_n$ such that there exists states s_0, \dots, s_n, s_{n+1} with $s_i \xrightarrow{\delta_i} \xrightarrow{\theta_i} s_{i+1}$ for all $i \in [0, n)$ and $s_n \xrightarrow{\delta_n} s_{n+1}$. We say that a STT is feasible if it has at least one realization.

Property Specification. We focus on the analysis of timed safety properties, which we characterize by an invariant formula that has to hold for all reachable states of a TA. These properties state, for instance, that there are certain locations in which the value of a clock variable is not above, equal to or below a certain (integer) bound. Formally, let $T = (L, l^0, C, \Sigma, \Theta, I)$ be a TA. A *timed safety property* Π is a Boolean combination of atomic clock constraints and *location predicates* $@l$ where $l \in L$. A location predicate $@l$ holds in a state (l', u) of T iff $l' = l$. We say that a STT S witnesses a violation of Π in T if there exists a realization of S whose induced final state does not satisfy Π . We refer to such an STT as a *timed diagnostic trace* of T for Π .

T satisfies Π iff all its reachable states satisfy Π . This problem can be decided using model checking tools such as Kronos [31] and UPPAAL [6]. UPPAAL in particular computes a finite abstraction of the state space of an NTA using a zone graph construction. Reachability analysis is then performed by an on-the-fly search of the zone graph. If the property is violated, the tool generates a feasible TDT that witnesses the violation. The objective of our work is to analyze TDTs and to propose repairs for the property violation that they represent. We use TDTs generated by the UPPAAL tool in our implementation, but we maintain that our results can be adapted to any other tool producing TDTs.

We further note that UPPAAL takes a *network of timed automata* (NTA) as input, which is a CCS [24] style parallel composition of timed automata $T_1 \mid \dots \mid T_n$. Since our analysis and repair techniques focus on timing-related errors rather than synchronization errors, we use TAs rather than NTAs in our formalization. However, our implementation works on NTAs.

Example 1. The running example that we use throughout the paper consists of an NTA of two timed automata, depicted in Fig. 2. As alluded to in the introduction, the TAs `dbServer` and `db` synchronize via the exchange of messages modeled by the pairs of send and receive actions `req!` and `req?`, respectively, `ser!` and `ser?`. The transmission time of the `req` message is controlled by the clock variable `x` and can range between 1 and 2 time units. This is achieved by the location invariant $x \leq 2$ on the `reqReceived` location in `db` together with the transition guard $x \geq 1$ on the transition from `reqReceived` to `reqProcessing`. A similar mechanism using clock variable `z` is used to constrain the timing of the transfer of message `ser` to be within 1 and 2 time units. The processing time in `dbServer` is constrained to exactly 1 time unit by the location invariant $y \leq 1$ and the transition guard $y \geq 1$. In `dbServer`, a transition to location `timeout` can be triggered when the guard $z = 2$ is satisfied in location `serReceiving`. The clock variable `x`, which is not reset until the next `req` message is sent, is recording the time that has elapsed since sending `req` and is used in location `serReceiving` in order to verify if more than 4 time units have passed since `req` was sent. The timed safety property that we will consider for our example is $\Pi = \neg @\text{dbServer}.\text{serReceiving} \vee (x < 4)$. For the violation of this property, UPPAAL produces the TDT $S = \theta_0 \dots \theta_3$ where

$$\begin{aligned}\theta_0 &= ((\text{initial}, \text{reqAwaiting}), \emptyset, \tau, \emptyset, (\text{reqCreate}, \text{reqAwaiting})) \\ \theta_1 &= ((\text{reqCreate}, \text{reqAwaiting}), \emptyset, \tau, \{x\}, (\text{reqSent}, \text{reqReceived})) \\ \theta_2 &= ((\text{reqSent}, \text{reqReceived}), \{x \geq 1\}, \tau, \{y\}, (\text{reqSent}, \text{reqProc.})) \\ \theta_3 &= ((\text{reqSent}, \text{reqProc.}), \{y \geq 1\}, \tau, \{z\}, (\text{serReceiving}, \text{reqAwait.})).\end{aligned}$$

3 Logical Encoding of Timed Diagnostic Traces

Our analysis relies on a logical encoding of TDTs in the theory of quantifier-free linear real arithmetic. For the remainder of this paper, we fix a TA $T = (L, l^0, C, \Sigma, \Theta, I)$ with a safety property Π and assume that $S = \theta_0, \dots, \theta_{n-1}$ is an STT of T . We use the following notation for our logical encoding where $j \in [0, n+1]$ is a position in a realization of S and $c \in C$ is a clock:

- l_j denotes the location of the pre state of θ_j for $j < n$ and the location of the post state of θ_{j-1} for $j = n$.
- c_j denotes the value of clock variable c when reaching the state at position j .
- δ_j denotes the delay of the delay transition leaving the state at position $j \leq n$.
- reset_j denotes the set of clock variables that are being reset by action θ_j for $j < n$.
- $\text{ibounds}(c, l)$ denotes the set of pairs (β, \sim) such that the atomic clock constraint $c \sim \beta$ appears in the location invariant $I(l)$.

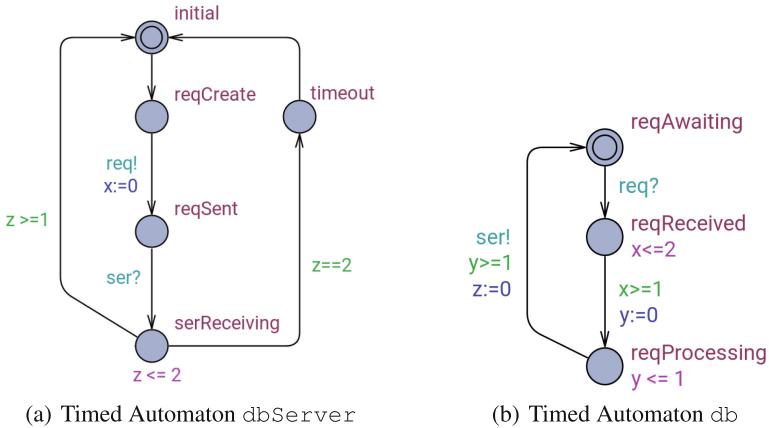


Fig. 2. Network of timed automata - running example

- $gbounds(c, \theta)$ denotes the set of pairs (β, \sim) such that the atomic clock constraint $c \sim \beta$ appears in the guard of action θ .

To illustrate the use of *ibounds*, assume location l to be labeled with invariants $x > 2 \wedge x \leq 4 \wedge y \leq 1$, then $ibounds(x, l) = \{(2, >), (4, \leq)\}$. The usage of *gbounds* is accordingly.

Definition 2. The timed diagnostic trace constraint system associated with STT S is the conjunction T of the following constraints:

$$\begin{aligned}
 \mathcal{C}_0 &\equiv \bigwedge_{c \in C} c_0 = 0 && \text{(clock initialization)} \\
 \mathcal{A} &\equiv \bigwedge_{j \in [0, n]} \delta_j \geq 0 && \text{(time advancement)} \\
 \mathcal{R} &\equiv \bigwedge_{c \in reset_j} c_{j+1} = 0 && \text{(clock resets)} \\
 \mathcal{D} &\equiv \bigwedge_{c \notin reset_j} c_{j+1} = c_j + \delta_j && \text{(sojourn time)} \\
 \mathcal{I} &\equiv \bigwedge_{(\beta, \sim) \in ibounds(c, l_j)} c_j \sim \beta \wedge c_j + \delta_j \sim \beta && \text{(location invariants)} \\
 \mathcal{G} &\equiv \bigwedge_{(\beta, \sim) \in gbounds(c, \theta_j)} c_j + \delta_j \sim \beta && \text{(transition guards)} \\
 \mathcal{L} &\equiv @l_n \wedge \bigwedge_{l \neq l_n} \neg @l && \text{(location predicates)}
 \end{aligned}$$

Let further $\Phi \equiv \Pi[\mathbf{c}_{n+1}/\mathbf{c}]$ where $\Pi[\mathbf{c}_{n+1}/\mathbf{c}]$ is obtained from Π by substituting all occurrences of clocks $c \in C$ with c_{n+1} . Then the Π -extended TDT constraint system associated with S is defined as $T^\Pi = T \wedge \neg\Phi$.

To illustrate the encoding consider the transition Θ_3 of the TDT in Example 1 corresponding to the transition from state (`reqSent`, `reqProcessing`) to state (`serReceiving`, `reqAwaiting`) while resetting clock z in the NTA of Fig. 2. The encoding for the constraints on the clocks x , y and z is as following: $y_3 + d_3 \geq 1$, $z_4 = 0$, $x_4 = x_3 + d_3$ and $y_4 = y_3 + d_3$.

Lemma 1. $\delta_0^c, \dots, \delta_n^c$ is a realization of an STT S iff there exists a satisfying variable assignment ι for T such that for all $j \in [0, n]$, $\iota(\delta_j) = \delta_j^c$.

Theorem 1. An STT S witnesses a violation of Π in T iff T^Π is satisfiable.

4 Repair

We propose a repair technique that analyzes the responsibility of clock bound values occurring in a single TDT for causing the violation of a specification Π . The analysis suggests possible syntactic repairs. In a second step we define an admissibility test that assesses the admissibility of the repair in the context of the complete TA model. Throughout this section, we assume that S is a TDT for T and Π .

Clock Bound Variation. We introduce *bound variation variables* v that stand for *correction values* that the repair will add to the clock bounds occurring in location invariants and transition guards. The values are chosen such that none of the realizations of S in the modified automaton still witnesses a violation of Π . This is done by defining a new constraint system that captures the conditions on the variable v under which the violation of Π will not occur in the corresponding trace of the modified automaton. Using this constraint system, we then define a maximum satisfiability problem whose solution minimizes the number of changes to T that are needed to achieve the repair.

Recall that the clock bounds occurring in location invariants and in transition guards are represented by the *ibounds* and *gbounds* sets defined for the TDT S . Notice that each clock variable c may be associated with $m_{c,l}$ different clock bounds in the location invariant of l , denoted by the set $ibounds(c, l) = \{(\beta_1^{c,l}, \sim_1^{c,l}), \dots, (\beta_{m_{c,l}}^{c,l}, \sim_{m_{c,l}}^{c,l})\}$. Similarly, we enumerate the bounds in $gbounds(c, \theta)$ as $(\beta_k^{c,\theta}, \sim_k^{c,\theta})$. To reduce notational clutter, we let the meta variable r stand for the pairs of the form c, l or c, θ . We then introduce bound variation variables v_k^r describing the possible static variation in the TA code for the clock bound β_k^r and modify the TDT constraint system accordingly. A variation of the bounds only affects the location invariant constraints \mathcal{I} and the transition guard constraints \mathcal{G} . We thus define an appropriate invariant variation constraint \mathcal{I}^{bv} and guard variation constraint \mathcal{G}^{bv} that capture the clock bound modifications:

$$\mathcal{I}^{bv} \equiv \bigwedge_{(\beta_k^r, \sim_k^r) \in ibounds(c, l_j)} c_j \sim_k^r (\beta_k^r + v_k^r) \wedge c_j + \delta_j \sim_k^r (\beta_k^r + v_k^r)$$

$$\mathcal{G}^{bv} \equiv \bigwedge_{(\beta_k^r, \sim_k^r) \in gbounds(c, \theta_j)} c_j + \delta_j \sim_k^r (\beta_k^r + v_k^r)$$

We also need constraints ensuring that the modified clock bounds remain positive:

$$\mathcal{Z}^{bv} \equiv \bigwedge_{(\beta_k^r, \sim_k^r) \in ibounds(c, l_j) \cup gbounds(c, \theta_j)} \beta_k^r + v_k^r \geq 0$$

Putting all of this together we obtain the *bound variation TDT constraint system*

$$\mathcal{T}^{bv} \equiv \mathcal{C}_0 \wedge \mathcal{A} \wedge \mathcal{R} \wedge \mathcal{D} \wedge \mathcal{I}^{bv} \wedge \mathcal{G}^{bv} \wedge \mathcal{Z}^{bv} \wedge \mathcal{L}$$

which captures all realizations of S in TAs T^{bv} that are obtained from T by modifying the clock bounds β_k^r by some semantically consistent variations v_k^r .

Consider the bound variation for the guard $y \geq 1$ of transition Θ_3 in Example 1. The modified guard constraint, a conjunct in \mathcal{G}^{bv} , is $y_3 + d_3 \geq 1 + v_3^y$. The corresponding non-negativity constraint from \mathcal{Z}^{bv} is $1 + v_3^y \geq 0$.

Repair by Bound Variation Analysis. The objective of the bound variation analysis is to provide hints to the system designer regarding which minimal syntactic changes to the considered model might prevent the violation of property Π . Minimality here is considered with respect to the number of clock bound values in invariants and guards that need to be changed.

We implement this analysis by using the bound variation TDT constraint system \mathcal{T}^{bv} to derive an instance of the partial MaxSMT problem whose solutions yield candidate repairs for the timed automaton T . The partial MaxSMT problem takes as input a finite set of assertion formulas belonging to a fixed first-order theory. These assertions are partitioned into *hard* and *soft* assertions. The hard assertions \mathcal{F}_H are assumed to hold and the goal is to find a maximizing subset $\mathcal{F}' \subseteq \mathcal{F}_S$ of the soft assertions such that $\mathcal{F}' \cup \mathcal{F}_H$ is satisfiable in the given theory.

For our analysis, the hard assertions consist of the conjunction

$$\mathcal{F}_H^{bv} \equiv (\exists \delta_j, c_j. \mathcal{T}^{bv}) \wedge (\forall \delta_j, c_j. \mathcal{T}^{bv} \Rightarrow \Phi).$$

Note that the free variables of \mathcal{F}_H^{bv} are exactly the bound variation variables v_k^r . Given a satisfying assignment ι for \mathcal{F}_H^{bv} , let T_ι be the timed automaton obtained from T by adding to each clock bound β_k^r the according variation value $\iota(v_k^r)$ and let S_ι be the TDT corresponding to S in T_ι . Then \mathcal{F}_H^{bv} guarantees that

1. S_ι is feasible, and
2. S_ι has no realization that witnesses a violation of Π in T_ι .

We refer to such an assignment ι as a *local clock bound repair* for T and S . To obtain a minimal local clock bound repair, we use the soft assertions given by the conjunction

$$\mathcal{F}_S^{bv} \equiv \bigwedge_{(\beta_k^r, \sim_k^r) \in ibounds(c, l_j) \cup gbounds(c, \theta_j)} v_k^r = 0.$$

Clearly $\mathcal{F}_H^{bv} \wedge \mathcal{F}_S^{bv}$ is unsatisfiable because $\mathcal{T}^{bv} \wedge \mathcal{F}_S^{bv}$ is equisatisfiable with \mathcal{T} , and $\mathcal{T} \wedge \neg\Phi$ is satisfiable by assumption. However, if there exists at least one local clock

bound repair for T and S , then \mathcal{F}_H^{bv} alone is satisfiable. In this case, the MaxSMT instance $\mathcal{F}_H^{bv} \cup \mathcal{F}_S^{bv}$ has at least one solution. Every satisfying assignment of such a solution corresponds to a local repair that minimizes the number of clock bounds that need to be changed in T .

Note that hard and soft assertions remain within a decidable logic. Using an SMT solver such as Z3, we can enumerate all the optimal solutions for the partial MaxSMT instance and obtain a minimal local clock bound repair from each of them.

Example 2. We have applied the bound variation repair analysis to the TDT from Example 1, using TARTAR, which calls Z3. The following repairs were computed:

1. $v_1^{z,l_5} = -1$. This corresponds to a variation of the location invariant regarding clock z in location 5 of the TDT, corresponding to location `dbServer.serReceiving`, to read $z \leq 1$ instead of $z \leq 2$. This indicates that the violation of the bound on the total duration of the transaction, as indicated by a return to the `serReceiving` location and a value greater than 4 for clock x , can be avoided by ensuring that the time taken for transmitting the `ser` message to the `dbServer` is constrained to take exactly 1 time unit.
2. A further computed repair is $v_1^{x,l_2} = -1$. Interpreting this variation in the context of Example 1 means that location `db.reqReceived` will be left when the clock x has value 1. In other words, the transmission of the message `req` to the `db` takes exactly one time unit, not between 1 and 2 time units as in the unrepaired model.
3. Another possible repair implies the modification of two clock bounds. This is no longer an optimal solution and no further optimal solution exists. Notice that even non-optimal solutions might provide helpful insight for the designer, for instance if optimal repairs turn out not to be implementable, inadmissible or leading to a property violation. It is therefore meaningful to allow a practical tool implementation to compute more than just the optimal repairs.

5 Admissibility of Repair

The synthesized repairs that lead to a TA T_r change the original TA T in fundamental ways, both syntactically and semantically. This brings up the question whether the synthesized repairs are admissible. In fact, one of the key questions is what notion of admissibility is meaningful in this context.

A *timed trace* [7] is a sequence of timed actions $\xi = (t_1, a_1), (t_2, a_2), \dots$ that is generated by a run of a TA, where $t_i \leq t_{i+1}$ for all $i \geq 1$. The timed language for a TA T is the set of all its timed traces, which we denote by $\mathcal{L}_T(T)$. The untimed language of T consists of words over T 's alphabet Σ so that there exists at least one timed trace of T forming this word. Formally, for a timed trace $\xi = (t_1, a_1), (t_2, a_2) \dots$, the untimed operator $\mu(\xi)$ returns an untimed trace $\xi_\mu = a_1 a_2 \dots$. We define the untimed language $\mathcal{L}_\mu(T)$ of the TA T as $\mathcal{L}_\mu(T) = \{\mu(\xi) \mid \xi \in \mathcal{L}_T(T)\}$.

Let B be a Büchi automaton (BA) [10] over some alphabet Σ . We write $\mathcal{L}(B) \subseteq \Sigma^\omega$ for the language accepted by B . Similarly, we denote by $\mathcal{L}_f(B) \subseteq \Sigma^*$ the language accepted by B if it is interpreted as a nondeterministic finite automaton (NFA). Further, we write $\text{pref}(\mathcal{L}(B))$ to denote the set of all finite prefixes of words in $\mathcal{L}(B)$.

For a given NFA or BA M , the *closure* $c1(M)$ denotes the automaton obtained from M by turning all of its states into accepting states. We call M closed iff $M = c1(M)$. Notice that a Büchi automaton accepts a safety language if and only if it is closed [1].

Admissibility Criteria. From a *syntactic* point of view the repair obtained from a satisfying assignment ι of the MaxSMT instance ensures that T_ι is a syntactically valid TA model by, for instance, placing non-negativity constraints on repaired clock bounds. In case repairs alter right hand sides of clock constraints to rational numbers, this can easily be fixed by normalizing all clock constraints in the TA.

From a *semantic* perspective, the impact of the repairs is more profound. Since the repairs affect time bounds in location invariants and transition guards, as well as clock resets, the behavior of T_ι may be fundamentally different from the behavior of T .

- First, the computed repair for one property Π may render another property Π' violated. To check admissibility of the synthesized repair with respect to the set of all properties $\hat{\Pi}$ in the system specification, a full re-checking of $\hat{\Pi}$ is necessary.
- Second, a repair may have introduced zenoness and timelock [4] into T_ι . As discussed in [4], there exists both an over-approximating static test for zenoness as well as a model checking based precise test for timelocks that can be used to verify whether the repair is admissible in this regard.
- Third, due to changes in the possible assignment of time values to clocks, reachable locations in the TA T may become unreachable in T_ι , and vice versa. On the one hand, this means that some functionalities of the system may no longer be provided since part of the actions in T will no longer be executable in T_ι , and vice versa. Further, a reduction in the set of reachable locations in T_ι compared to T may mean that certain locations with property violations in T are no longer reachable in T_ι , which implies that certain property violations are masked by a repair instead of being fixed. On the other hand, the repair leading to locations becoming reachable in T_ι that were unreachable in T may have the effect that previously unobserved property violations become visible and that T_ι possesses functionality that T does not have, which may or may not be desirable.

It should be pointed out that we assess admissibility of a repair leading to T_ι with respect to a given TA model T , and not with respect to a correct TA model T^* satisfying Π .

Functional Equivalence. While various variants of semantic admissibility may be considered, we are focusing on a notion of admissibility that ensures that a repair does not unduly change the functional behavior of the modeled system while adhering to the timing constraints of the repaired system. We refer to this as *functional equivalence*. The functional capabilities of a timed system manifest themselves in the sets of action or transition traces that the system can execute. For TAs T and T_ι this means that we need to consider the languages over the action or transition alphabets that these TAs define. Considering the timed languages of T and T_ι , we can state that $\mathcal{L}_T(T) \neq \mathcal{L}_T(T_\iota)$ since the repair forces at least one timed trace to be purged from $\mathcal{L}_T(T)$. This means that equivalence of the timed languages cannot be an admissibility criterion ensuring functional equivalence. At the other end of the spectrum we may relate the de-timed

languages of T and T_ι . The *de-time* operator $\alpha(T)$ is defined such that it omits all timing constraints and resets from any TA T . Requiring $\mathcal{L}(\alpha(T)) = \mathcal{L}(\alpha(T_\iota))$ is tempting since it states that when eliminating all timing related features from T and from the repaired T_ι , the resulting action languages will be identical.

However, this admissibility criterion would be flawed, since the repair in T_ι may imply that unreachable locations in T will be reachable in T_ι , and vice versa. This may have an impact on the untimed languages, and even though $\mathcal{L}(\alpha(T)) = \mathcal{L}(\alpha(T_\iota))$ it may be that $\mathcal{L}_\mu(T) \neq \mathcal{L}_\mu(T_\iota)$. To illustrate this point, consider the running example in Fig. 2 and assume the invariant in location `dbServer.reqReceiving` to be modified from $z \leq 2$ to $z \leq 1$ in the repaired TA T_ι . Applying the de-time operator to T_ι implies that the location `dbServer.timeout`, which is unreachable in T_ι , becomes reachable in the de-timed model. Since `dbServer.timeout` is reachable in T , the TA T and T_ι are not functionally equivalent, even though their de-timed languages are identical. Notice that for the untimed languages $\mathcal{L}_\mu(T) \neq \mathcal{L}_\mu(T_\iota)$ holds since no timed trace in $\mathcal{L}_T(T_\iota)$ reaches location `timeout`, even though such a timed trace exists in $\mathcal{L}_T(T)$. In detail, $\mathcal{L}_\mu(T)$ contains the untimed trace $\Theta_0\Theta_1\Theta_2\Theta_3\Theta_4$ that is missing in $\mathcal{L}_\mu(T_\iota)$ and where Θ_4 is the transition towards the location `dbServer.timeout`. As consequence, we resort to considering the untimed languages of T and T_ι and require $\mathcal{L}_\mu(T) = \mathcal{L}_\mu(T_\iota)$. It is easy to see that $\mathcal{L}_\mu(T) = \mathcal{L}_\mu(T_\iota) \Rightarrow \mathcal{L}(\alpha(T)) = \mathcal{L}(\alpha(T_\iota))$. In other words, the equivalence of the untimed languages ensures functional equivalence.

Admissibility Test. Designing an algorithmic admissibility test for functional equivalence is challenging due to the computational complexity of determining the equivalence of the untimed languages $\mathcal{L}_\mu(T)$ and $\mathcal{L}_\mu(T_\iota)$. While language equivalence is decidable for languages defined by Büchi Automata, it is undecidable for timed languages [3]. For untimed languages, however, this problem is again decidable [3]. The algorithmic implementation of the test for functional equivalence that we propose proceeds in two steps.

- First, the untimed languages $\mathcal{L}_\mu(T)$ and $\mathcal{L}_\mu(T_\iota)$ are constructed. This requires an untim transformation of T and T_ι yielding Büchi automata representing $\mathcal{L}_\mu(T)$ and $\mathcal{L}_\mu(T_\iota)$. While the standard untim transformation for TAs [3] relies on a region construction, we propose a transformation that relies on a zone construction [14]. This will provide a more succinct representation of the resulting untimed languages and, hence, a more efficient equivalence test.
- Second, it needs to be determined whether $\mathcal{L}_\mu(T) = \mathcal{L}_\mu(T_\iota)$. As we shall see, the obtained Büchi automata are closed. Hence, we can reduce the equivalence problem for these ω -regular languages to checking equivalence of the regular languages obtained by taking the finite prefixes of the traces in $\mathcal{L}_\mu(T)$ and $\mathcal{L}_\mu(T_\iota)$. This allows us to interpret the Büchi automata obtained in the first step as NFAs, for which the language equivalence check is a standard construction [15].

Automata for Untimed Languages. The construction of an automaton representing an untimed language, here referred to as an *untim construction*, has so far been proposed based on a region abstraction [3]. The region abstraction is known to be relatively inefficient since the number of regions is, among other things, exponential in the number of

clocks [4]. We therefore propose an untimed construction based on the construction of a zone automaton [14] which in the worst case is of the same complexity as the region automaton, but on the average is more succinct [7].

Definition 3 (Untimed Büchi Automaton). Assume a TA T and the corresponding zone automaton $\llbracket T \rrbracket_Z = (S_Z, s_Z^0, \Sigma_Z, \Theta_Z)$. We define the untimed Büchi automaton as the closed BA $B_T = (S, \Sigma, \rightarrow, S_0, F)$ obtained from $\llbracket T \rrbracket_Z$ such that $S = S_Z$, $\Sigma = \Sigma_Z \setminus \{\delta\}$ and $S_0 = \{s_Z^0\}$. For every transition in Θ_Z with a label $a \in \Sigma$ we add a transition to \rightarrow created by the rule $\frac{(l, z) \xrightarrow{\delta} (l, z^\uparrow) \xrightarrow{a} (l', z')}{(l, z) \xrightarrow{a} (l', z')}$ with $z^\uparrow = \{v + d \mid v \in z, d \in \mathbb{R}_{\geq 0}\}$. In addition, we add self-transitions $(l, z) \xrightarrow{\tau} (l, z)$ to every state $(l, z) \in S_B$.

The following observations justify this definition:

- A timed trace of T may remain forever in the same location after a finite number of action transitions. In order to enable B to accept this trace, we add a self-transition labeled with τ to \rightarrow for each state $s \in S$ in B_T , and later define s as accepting. These τ -self-transitions extend every finite timed trace t leading to a state in S_τ to an infinite trace $t.\tau^\omega$.
- The construction of the acceptance set F is more intricate. Convergent traces are often excluded from consideration in real-time model checking [4]. As a consequence, in the untimed construction proposed in [3], only a subset of the states in S may be included in F . A repair may render a subgraph of the location graph of T that is only reachable by divergent traces, into a subgraph in T_ℓ that is only reachable by convergent traces. However, excluding convergent traces is only meaningful when considering unbounded liveness properties, but not when analyzing timed safety properties, which in effect are safety properties. As argued in [7], unbounded liveness properties appear to be less important than timed safety properties in timed systems. This is due to the observation that divergent traces reflect unrealistic behavior in the limit, but finite prefixes of infinite divergent traces, which only need to be considered for timed safety properties, correspond to realistic behavior. This observation is also reflected in the way in which, e.g., UPPAAL treats reachability by convergent traces. In conclusion, this justifies our choice to define the zone automaton in the untimed construction as a closed BA, i.e., $F = S$.

Theorem 2 (Correctness of Untimed Büchi Automaton Construction). For an untimed Büchi automaton B_T derived from a TA T according to Definition 3 it holds that $\mathcal{L}(B_T) = \mathcal{L}_\mu(T)$.

Equivalence Check for Untimed Languages. Given that the zone automaton construction delivers closed BAs we can reduce the admissibility test $\mathcal{L}_\mu(T) = \mathcal{L}_\mu(T_\ell)$ defined over infinite languages to an equivalence test over the finite prefixes of these languages, represented by interpreting the zone automata as NFAs. The following theorem justifies this reduction.

Theorem 3 (Language Equivalence of Closed BA). Given closed Büchi automata B and B' , if $\mathcal{L}_f(B) = \mathcal{L}_f(B')$ then $\mathcal{L}(B) = \mathcal{L}(B')$.

Discussion. One may want to adapt the admissibility test so that it only considers divergent traces, e.g., in cases where only unbounded liveness properties need to be preserved by a repair. This can be accomplished as follows. First, an overapproximating non-zoneness test [4] can be applied to T and T_L . If it shows non-zoneness, then one knows that the respective TA does not include convergent traces. If this test fails, a more expensive test needs to be developed. It requires a construction of the untimed Büchi automata using the approach from [3], and subsequently a language equivalence test of the untimed languages accepted by the untimed BAs using, for instance, the automata-theoretic constructions proposed in [9].

6 Case Studies and Experimental Evaluation

We have implemented the repair computation and admissibility test in a proof-of-concept tool called TARTAR. We present the architecture of TARTAR and then evaluate the proposed method by applying TARTAR to several case studies.

Tool Architecture. The control loop of TARTAR, depicted in Fig. 3, computes repairs for a given UPPAAL model and a given property Π using the following steps:

1. *Counterexample Creation.* TARTAR calls UPPAAL with parameters to compute and store a shortest symbolic TDT in XML format, in case Π is violated.
2. *Diagnostic Trace Creation.* Parsing the model and the TDT, TARTAR creates $\mathcal{F}_H^{bv} \wedge \mathcal{F}_S^{bv}$ as defined in Sect. 4. Z3 can only solve the MaxSMT problem for quantifier-free linear real arithmetic. Hence, TARTAR first performs a quantifier elimination on the constraints $\forall \delta_j, c_j. \mathcal{T}^{bv} \Rightarrow \Phi$ of \mathcal{F}_H^{bv} .
3. *Repair Computation.* Next, TARTAR attempts to compute a repair, by using Z3 to solve the generated quantifier-free MaxSMT instance. In case no solution is found, TARTAR terminates. Otherwise, TARTAR returns the repair that has been computed from the model of the MaxSMT solution.
4. *Admissibility Check.* Using adapted routines provided by the opaal model checker [11], TARTAR checks the admissibility of the computed repair. To do so, TARTAR modifies the constraints of the considered UPPAAL model as indicated by the computed repair. It calls opaal in order to compute the timed transition system (TTS) of the original and the repaired UPPAAL model. TARTAR then checks whether the two TTS have equivalent untimed languages, in which case the repair is admissible. This check is implemented using the library AutomataLib included in the package LearnLib [16].
5. *Iteration.* TARTAR is designed to enumerate all repairs, starting with the minimal ones, in an iterative loop. To accomplish this, at the end of each iteration i a new \mathcal{V}_{i+1}^{bv} is generated by forcing the bound variation variables that were used in the i -th repair to 0. This excludes the repair computed in iteration i from further consideration. Using \mathcal{V}_{i+1}^{bv} , TARTAR iterates back to Step 3 to compute another repair.

Evaluation Strategy. The evaluation of our analysis is based on ideas taken from mutation testing [18]. Mutation testing evaluates a test set by systematically modifying the program code to be tested and computing the ratio of modifications that are detected by the test set. Real-time system models that contain violations of timed safety properties are not available in significant numbers. We therefore need to seed faults in existing models and check whether those can be found by our automated repair. An objective of mutation testing is that testing a proportion of the possible modification yields satisfactory results [18]. In order to evaluate repairs for erroneous clock bounds in invariants and transition guards we seed modifications to all bounds of clock constraints by the amount of $\{-10, -1, +1, +0.1 \cdot M, +M\}$, where M is the maximal bound a clock is compared against in a given model. If a thus seeded modification leads to a syntactically invalid UPPAAL model, then UPPAAL returns an exception and we ignore this modification. In analogy to mutation testing, we compute the count of TDTs for which our analysis finds an admissible repair.

Experiments. We have applied this modification seeding strategy to eight UPPAAL models (see Table 1). Not all of the models that we considered have been published with a property that can be violated by mutating a clock constraint. For those models, we suggest a suitable timed safety property specifying an invariant condition. In particular, we add a property to the Bando [29] model which ensures that, for as long as the sender is active, its clock never exceeds the value of 28,116 time units. In the FDDI token ring protocol [29], the property that we use checks whether the first member of the ring never remains for more than 140 time units in any given state. The Viking model is taken from the set of test models of opaal [26]. For this model we use a property that checks whether one of the Viking processes can only enter a safe state during the first 60 time units. Note that all of these properties are satisfied by the unmodified models.

The results of the clock bound repair computed by TARTAR for all considered models are summarized in Table 1. The seeded modifications are characterized quantitatively by the count #Seed of analyzed modified models, the count #TDT of modified models that return a TDT for the considered property, the maximal time T_{UP} UPPAAL needs to create a TDT per analyzed model, and the length Len. of the longest TDT found. For the computation of a repair we give the count #Rep. of all repairs that were computed, the count #Adm. of computed admissible repairs, the count of TDTs #Sol. for which an admissible repair was found, the maximal time T_{QE} that the quantifier elimination required, the average time effort T_R to compute a repair, the standard deviation SD_R for the computation time of a repair, the time effort T_{Adm} for an admissibility check, the maximal count of variables #Var, and the maximal count of constraints #Con. used in \mathcal{V}_{i+1}^{bv} . The maximal memory consumption was at most 17MB for the repair analysis and 478MB for the admissibility test. We performed all experiments on a computer with an i7-6700K CPU (4.0GHz), 60 GB of RAM and a Linux operating system.

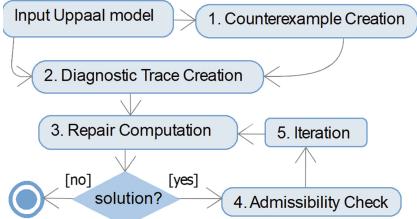


Fig. 3. Control loop of TARTAR

We found 60 TDTs by seeding violations of the timed safety property and TARTAR returned 204 repairs for these TDTs. TARTAR proposed an admissible repair for 55 (91%) TDTs and at least one repair for 57 (95%) TDTs. For 3 out of the total of 14 TDTs found for the SBR model no repair was computed since the timeout of the quantifier elimination was reached after 2 minutes. For all other models, no timeout occurred.

Space limitations do not permit us to describe all models and computed repairs in detail, we therefore focus on the pacemaker case study. One of the modification increases a location invariant of this model that controls the minimal heart period from 400 to 1,600. The modification allows the pacemaker to delay an induced ventricular beat for too long so that this violates the property that the time between two ventricular beats of a heart is never longer than the maximal heart period of 1,000. TARTAR finds three repairs. Two repairs reduce the maximal time delay between two ventricular or articular heart beats of the patient. The repairs are classified as inadmissible. In the model context this appears to be reasonable since the repairs would restrict the environment of the pacemaker, and not the pacemaker itself. The third repair is admissible and reduces the bound modified during the seeding of bound modifications by 600.5. The minimal heart period is then below or equal to the maximal heart period of 1,000.

Result Interpretation. Our repair strategy minimizes the number of repairs but does not optimize the computed value. For instance, in the pacemaker model the computed repair of 600.5 would be a correct and admissible repair even if the value was reduced to 600, which would be the minimal possible repair value.

A comparison of the values T_{QE} and T_R reveals that, perhaps unsurprisingly, the quantifier elimination step is computationally almost an order of magnitude more expensive than the repair computation. Overall, the computational cost ($T_{QE} + T_R$) correlates with the number of variables in the constraint system, which depends in turn on the length of the TDT and the number of clocks referenced along the TDT. Consider, for instance, that the pacemaker model has a TDT of maximal length 9 with 116 variables, and the repair requires 0.193 s and 2.070 MB. On the other hand, the Bando model produces a longer maximal TDT of length 279 with 1,156 variables and requires 6.555 s and 16.650 MB. The impact of the number of clock constraints and clock variables on the computation costs can be seen, for instance, in the data for the pacemaker and FDDI models. While the pacemaker model has a shorter TDT than the Viking model (9 vs. 18), the constraint counts (294 vs. 140) of the pacemaker model are higher than for

Table 1. Experimental results for clock bound repair computation using TARTAR

Model	# Seed	# TDT	T_{UP}	Len.	# Rep.	# Adm.	# Sol.	T_{QE}	T_R	SD_R	T_{Adm}	# Var.	# Con.
Repaired db Fig. 2	35	6	0.006 s	4	12	12	6	0.042 s	0.023 s	0.001	2.329 s	25	40
CSMA/CD [17]	90	6	0.012 s	2	36	16	6	0.020 s	0.021 s	0.000	3.060 s	16	36
Elevator [8]	35	3	0.004 s	1	6	6	3	0.071 s	0.028 s	0.005	2.374 s	6	16
Viking	85	3	0.009 s	18	6	6	3	0.032 s	0.042 s	0.002	2.821 s	120	140
Bando [29]	740	12	0.259 s	279	26	24	12	17.227 s	6.555 s	1.776	4.067 s	1,156	2,441
Pacemaker [19]	240	7	0.044 s	9	34	16	7	0.670 s	0.193 s	0.021	3.389 s	116	294
SBR [23]	65	14	0.066 s	81	42	26	9	20.776 s	2.568 s	0.441	34.120 s	256	410
FDDI [29]	100	9	0.025 s	5	42	30	9	0.046 s	0.029 s	0.001	2.493 s	59	93

the Viking model, which coincides with a higher computation time (0.193 s vs. 0.042 s) and a higher memory consumption (2.070 MB vs. 0.910 MB) compared to the Viking model.

We analyzed for every TDT the relationship between the length of the TDT and the computation time for a repair ($T_r = T_{QE} + T_R$), as well as the relationship between $\#Var$ and T_r by estimating Kendall's tau [13]. Kendall's tau is a measurement for the ordinal association between two measured quantities. A correlation is considered significant if the probability p that there is actually no correlation in a larger data set is below a certain threshold. The length of a TDT is significantly related ($\tau_1 = 0.673, p < .001$) to T_r . Also $\#Var$ is significantly related ($\tau_2 = 0.759, p < .001$) to T_r . $\#Var$ contains clocks for every step of a TDT, hence the combination of trace length and clock count tends to correlate higher than the trace length on its own. This supports our conjecture that the computation time of a repair depends on the trace length and the clock count.

The admissibility test appears to be quite efficient, with a maximum computation time of 34.120 s for the SBR model, which is one of the more complex models that were considered. We observed that most models were action-deterministic, which has a positive influence on the language equivalence test used during admissibility checking.

7 Conclusion

We have presented an approach to derive minimal repairs for timed reachability properties of TA and NTA models from TDTs in order to facilitate fault localization and debugging of such models during the design process. Our approach includes a formalization of TDTs using linear real arithmetic, a repair strategy based on MaxSMT solving, the definition of an admissibility criterion and test for the computed repairs, the development of a prototypical analysis and repair tool, and the application of the proposed method to a number of case studies of realistic complexity. To the best of our knowledge, this is the first rigorous treatment of counterexamples in real-time model checking. We are also not aware of any existing repair approaches for TA or NTA models. This makes a comparative experimental evaluation impossible. We have nonetheless observed that our analysis computes a significant number of admissible repairs within realistic computation time bounds and memory consumption.

Future research will address the development and implementation of repair strategies for further syntactic features in TAs and NTAs, including false comparison operators in invariants and guards, erroneous clock variable references, superfluous or missing resets for clocks, and wrong urgent state choices. We will furthermore address the interplay between different repairs and develop refined strategies to determine their admissibility. Finally, we plan to extend the approach developed in this paper to derive criteria for the actual causation of timing property violations in NTA models based on the counterfactual reasoning paradigm for causation.

Acknowledgments. We wish to thank Nikolaj Bjorner and Zvonimir Pavlinovic for advice on the use of Z3. We are grateful to Sarah Stoll for helping us with the statistical evaluation of the experimental results. This work is in part supported by the National Science Foundation (NSF) under grant CCF-1350574.

References

1. Alpern, B., Schneider, F.B.: Recognizing safety and liveness. *Distrib. Comput.* **2**(3), 117–126 (1987)
2. Alur, R., et al.: Syntax-guided synthesis. In: Dependable Software Systems Engineering, NATO Science for Peace and Security Series, D: Information and Communication Security, vol. 40, pp. 1–25. IOS Press (2015). <https://doi.org/10.3233/978-1-61499-495-4-1>
3. Alur, R., Dill, D.L.: A theory of timed automata. *Theor. Comput. Sci.* **126**(2), 183–235 (1994)
4. Baier, C., Katoen, J.P.: Principles of Model Checking. The MIT Press, Cambridge (2008)
5. Ben-Abdallah, H., Leue, S.: Timing constraints in message sequence chart specifications. In: FORTE. IFIP Conference Proceedings, vol. 107, pp. 91–106. Chapman & Hall (1997)
6. Bengtsson, J., Larsen, K., Larsson, F., Pettersson, P., Yi, W.: UPPAAL—a tool suite for automatic verification of real-time systems. In: Alur, R., Henzinger, T.A., Sontag, E.D. (eds.) HS 1995. LNCS, vol. 1066, pp. 232–243. Springer, Heidelberg (1996). <https://doi.org/10.1007/BFb0020949>
7. Bengtsson, J., Yi, W.: Timed automata: semantics, algorithms and tools. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) ACPN 2003. LNCS, vol. 3098, pp. 87–124. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27755-2_3
8. Tiage Brito: Uppaal elevator example (2015). <https://github.com/tfbrito/UPPAAL>. Accessed 20 Jan 2019
9. Clarke, E.M., Draghicescu, I.A., Kurshan, R.P.: A unified approach for showing language inclusion and equivalence between various types of omega-automata. *Inf. Process. Lett.* **46**(6), 301–308 (1993)
10. Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.): Handbook of Model Checking. Springer, Cham (2018)
11. Dalsgaard, A.E., et al.: A lattice model checker. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 487–493. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20398-5_37
12. Dierks, H., Kupferschmid, S., Larsen, K.G.: Automatic abstraction refinement for timed automata. In: Raskin, J.-F., Thiagarajan, P.S. (eds.) FORMATS 2007. LNCS, vol. 4763, pp. 114–129. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75454-1_10
13. Field, A.: Discovering Statistics Using IBM SPSS Statistics: and Sex and Drugs and Rock ‘n’ Roll. Sage, London (2013)
14. Henzinger, T.A., Nicollin, X., Sifakis, J., Yovine, S.: Symbolic model checking for real-time systems. *Inf. Comput.* **111**(2), 193–244 (1994)
15. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation, 2nd edn. Addison-Wesley, Stanford (2000)
16. Isberner, M., Howar, F., Steffen, B.: The open-source LearnLib. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 487–495. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_32
17. Jensen, H.E., Larsen, K.G., Skou, A.: Modelling and analysis of a collision avoidance protocol using spin and uppaal. In: The Spin Verification System. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 32, pp. 33–50. DIMACS/AMS (1996)
18. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. *IEEE Trans. Software Eng.* **37**(5), 649–678 (2011)
19. Jiang, Z., Pajic, M., Moarref, S., Alur, R., Mangharam, R.: Modeling and verification of a dual chamber implantable pacemaker. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 188–203. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28756-5_14

20. Jose, M., Majumdar, R.: Bug-assist: assisting fault localization in ANSI-C programs. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 504–509. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_40
21. Jose, M., Majumdar, R.: Cause clue clauses: error localization using maximum satisfiability. In: PLDI, pp. 437–446. ACM (2011)
22. Le, X.D., Chu, D., Lo, D., Le Goues, C., Visser, W.: S3: syntax- and semantic-guided repair synthesis via programming by examples. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, pp. 593–604. ACM (2017). <https://doi.org/10.1145/3106237.3106309>
23. Liu, S.: Analysing Timed Traces using SMT Solving. Master's thesis, University of Konstanz (2018)
24. Milner, R. (ed.): A Calculus of Communicating Systems. LNCS, vol. 92. Springer, Heidelberg (1980). <https://doi.org/10.1007/3-540-10235-3>
25. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
26. opaal: opaal test folder (2011). <http://opaal-modelchecker.com/opaal-ltsmin/>. Accessed 08 Nov 2018
27. Polsen, D.B., van Vliet, J.: Concrete Delays for Symbolic Traces. Master's thesis, Department of Computer Science, Aalborg University (2010). <https://projekter.aau.dk/projekter/files/3218338/report.pdf>
28. Reynolds, A., Kuncak, V., Tinelli, C., Barrett, C., Deters, M.: Refutation-based synthesis in SMT. Formal Methods in System Design (2017). <https://doi.org/10.1007/s10703-017-0270-2>
29. Uppaal: Uppaal benchmarks (2017). <http://www.it.uu.se/research/group/darts/uppaal/benchmarks/#benchmarks>. Accessed 20 Jan 2019
30. Yi, W., Pettersson, P., Daniels, M.: Automatic verification of real-time communicating systems by constraint-solving. In: FORTE. IFIP Conference Proceedings, vol. 6, pp. 243–258. Chapman & Hall (1994). <http://www.it.uu.se/research/group/darts/papers/texts/wpd-forfe94-full.pdf>
31. Yovine, S.: KRONOS: a verification tool for real-time systems. STTT **1**(1–2), 123–133 (1997)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Verifying Asynchronous Interactions via Communicating Session Automata

Julien Lange^{1(✉)} and Nobuko Yoshida²

¹ University of Kent, Canterbury, UK

j.s.lange@kent.ac.uk

² Imperial College London, London, UK



Abstract. This paper proposes a sound procedure to verify properties of communicating session automata (CSA), i.e., communicating automata that include multiparty session types. We introduce a new *asynchronous compatibility* property for CSA, called k -multiparty compatibility ($k\text{-MC}$), which is a strict superset of the synchronous multiparty compatibility used in theories and tools based on session types. It is decomposed into two bounded properties: (i) a condition called k -*safety* which guarantees that, within the bound, all sent messages can be received and each automaton can make a move; and (ii) a condition called k -*exhaustivity* which guarantees that all k -reachable send actions can be fired within the bound. We show that k -exhaustivity implies existential boundedness, and *soundly and completely* characterises systems where each automaton behaves equivalently under bounds greater than or equal to k . We show that checking $k\text{-MC}$ is PSPACE-complete, and demonstrate its scalability empirically over large systems (using partial order reduction).

1 Introduction

Communicating automata are a Turing-complete model of asynchronous interactions [10] that has become one of the most prominent for studying point-to-point communications over unbounded first-in-first-out channels. This paper focuses on a class of communicating automata, called *communicating session automata* (CSA), which strictly includes automata corresponding to *asynchronous multiparty session types* [28]. Session types originated as a typing discipline for the π -calculus [27, 66], where a session type dictates the behaviour of a process wrt. its communications. Session types and related theories have been applied to the verification and specification of concurrent and distributed systems through their integration in several mainstream programming languages, e.g., Haskell [44, 55], Erlang [49], F# [48], Go [11, 37, 38, 51], Java [30, 31, 34, 65], OCaml [56], C [52], Python [16, 47, 50], Rust [32], and Scala [61, 62]. Communicating automata and asynchronous multiparty session types [28] are closely related: the latter can be seen as a syntactical representation of the former [17] where a sending state corresponds to an internal choice and a receiving state to an external choice. This

© The Author(s) 2019

I. Dillig and S. Tasiran (Eds.): CAV 2019, LNCS 11561, pp. 97–117, 2019.

https://doi.org/10.1007/978-3-030-25540-4_6

correspondence between communicating automata and multiparty session types has become the foundation of many tools centred on session types, e.g., for generating communication API from multiparty session (global) types [30, 31, 48, 61], for detecting deadlocks in message-passing programs [51, 67], and for monitoring session-enabled programs [5, 16, 47, 49, 50]. These tools rely on a property called *multiparty compatibility* [6, 18, 39], which guarantees that communicating automata representing session types interact correctly, hence enabling the identification of correct protocols or the detection of errors in endpoint programs. Multiparty compatible communicating automata validate two essential requirements for session types frameworks: every message that is sent can be eventually received and each automaton can always eventually make a move. Thus, they satisfy the *abstract* safety invariant φ for session types from [63], a prerequisite for session type systems to guarantee safety of the typed processes. Unfortunately, multiparty compatibility suffers from a severe limitation: it requires that each execution of the system has a synchronous equivalent. Hence, it rules out many correct systems. Hereafter, we refer to this property as *synchronous multiparty compatibility* (SMC) and explain its main limitation with Example 1.

Example 1. The system in Fig. 1 contains an interaction pattern that is *not* supported by any definition of SMC [6, 18, 39]. It consists of a client (**c**), a server (**s**), and a logger (**l**), which communicate via unbounded FIFO channels. Transition $\mathbf{sr}!a$ denotes that sender puts (asynchronously) message a on channel **sr**; and transition $\mathbf{sr}?a$ denotes the consumption of a from channel **sr** by receiver. The client sends a *request* and some *data* in a fire-and-forget fashion, before waiting for a response from the server. Because of the presence of this simple pattern, the system cannot be executed synchronously (i.e., with the restriction that a send action can only be fired when a matching receive is enabled), hence it is rejected by all definitions of SMC from previous works, even though the system is safe (all sent messages are received and no automaton gets stuck).

Synchronous multiparty compatibility is reminiscent of a strong form of existential boundedness. Among the existing sub-classes of communicating automata (see [46] for a survey), existentially k -bounded communicating automata [22] stand out because they can be model-checked [8, 21] and they restrict the model in a natural way: any execution can be rescheduled such that the number of pending messages *that can be received* is bounded by k . However, existential boundedness is generally *undecidable* [22], even for a fixed bound k . This shortcoming makes it impossible to know when theoretical results are applicable.

To address the limitation of SMC and the shortcoming of existential boundedness, we propose a (decidable) sufficient condition for existential boundedness, called *k -exhaustivity*, which serves as a basis for a wider notion of new compatibility, called *k -multiparty compatibility* (k -MC) where $k \in \mathbb{N}_{>0}$ is a bound on the number of pending messages in each channel. A system is k -MC when it is (i) *k -exhaustive*, i.e., all k -reachable send actions are enabled within the bound, and (ii) *k -safe*, i.e., within the bound k , all sent messages can be received and each automaton can always eventually progress. For example, the system in Fig. 1 is k -multiparty compatible for any $k \in \mathbb{N}_{>0}$, hence it does not lead to communication

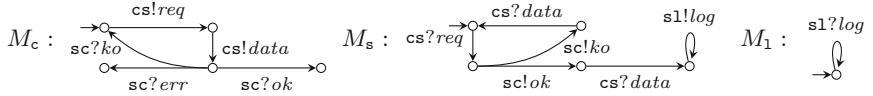


Fig. 1. Client-Server-Logger example.

errors, see Theorem 1. The k -MC condition is a natural constraint for real-world systems. Indeed any finite-state system is k -exhaustive (for k sufficiently large), while any system that is not k -exhaustive (resp. k -safe) for any k is unlikely to work correctly. Furthermore, we show that if a system of CSA validates k -exhaustivity, then each automaton locally behaves equivalently under any bound greater than or equal to k , a property that we call *local bound-agnosticity*. We give a *sound and complete* characterisation of k -exhaustivity for CSA in terms of local bound-agnosticity, see Theorem 3. Additionally, we show that the complexity of checking k -MC is PSPACE-complete (i.e., no higher than related algorithms) and we demonstrate empirically that its cost can be mitigated through (sound and complete) partial order reduction.

In this paper, we consider *communicating session automata* (CSA), which cover the most common form of asynchronous multiparty session types [15] (see Remark 3), and have been used as a basis to study properties and extensions of session types [6, 7, 18, 30, 31, 41, 42, 47, 49, 50]. More precisely, CSA are deterministic automata, whose every state is either sending (internal choice), receiving (external choice), or final. We focus on CSA that preserve the intent of internal and external choices from session types. In these CSA, whenever an automaton is in a sending state, it can fire any transition, no matter whether channels are bounded; when it is in a receiving state then at most one action must be enabled.

Synopsis. In Sect. 2, we give the necessary background on communicating automata and their properties, and introduce the notions of output/input bound independence which guarantee that internal/external choices are preserved in bounded semantics. In Sect. 3, we introduce the definition of k -multiparty compatibility (k -MC) and show that k -MC systems are safe for systems which validate the bound independence properties. In Sect. 4, we formally relate existential boundedness [22, 35], synchronisability [9], and k -exhaustivity. In Sect. 5 we present an implementation (using partial order reduction) and an experimental evaluation of our theory. We discuss related works in Sect. 6 and conclude in Sect. 7.

See [43] for a full version of this paper (including proofs and additional examples). Our implementation and benchmark data are available online [33].

2 Communicating Automata and Bound Independence

This section introduces notations and definitions of communicating automata (following [12, 39]), as well as the notion of output (resp. input) bound independence which enforces the intent of internal (resp. external) choice in CSA.

Fix a finite set \mathcal{P} of *participants* (ranged over by p, q, r, s , etc.) and a finite alphabet Σ . The set of *channels* is $\mathcal{C} \stackrel{\text{def}}{=} \{pq \mid p, q \in \mathcal{P} \text{ and } p \neq q\}$, $\mathcal{A} \stackrel{\text{def}}{=} \mathcal{C} \times \{!, ?\} \times \Sigma$ is the set of *actions* (ranged over by ℓ), Σ^* (resp. \mathcal{A}^*) is the set of finite words on Σ (resp. \mathcal{A}). Let w range over Σ^* , and ϕ, ψ range over \mathcal{A}^* . Also, ϵ ($\notin \Sigma \cup \mathcal{A}$) is the empty word, $|w|$ denotes the length of w , and $w \cdot w'$ is the concatenation of w and w' (these notations are overloaded for words in \mathcal{A}^*).

Definition 1 (Communicating automaton). A communicating automaton is a finite transition system given by a triple $M = (Q, q_0, \delta)$ where Q is a finite set of states, $q_0 \in Q$ is the initial state, and $\delta \subseteq Q \times \mathcal{A} \times Q$ is a set of transitions.

The transitions of a communicating automaton are labelled by actions in \mathcal{A} of the form $sr!a$, representing the *emission* of message a from participant s to r , or $sr?a$ representing the *reception* of a by r . Define $\text{subj}(pq!a) = \text{subj}(qp?a) = p$, $\text{obj}(pq!a) = \text{obj}(qp?a) = q$, and $\text{chan}(pq!a) = \text{chan}(qp?a) = pq$. The projection of ℓ onto p is defined as $\pi_p(\ell) = \ell$ if $\text{subj}(\ell) = p$ and $\pi_p(\ell) = \epsilon$ otherwise. Let \dagger range over $\{!, ?\}$, we define: $\pi_{pq}^\dagger(pq\dagger a) = a$ and $\pi_{pq}^{\dagger'}(sr\dagger a) = \epsilon$ if either $pq \neq sr$ or $\dagger \neq \dagger'$. We extend these definitions to sequences of actions in the natural way.

A state $q \in Q$ with no outgoing transition is *final*; q is *sending* (resp. *receiving*) if it is not final and all its outgoing transitions are labelled by send (resp. receive) actions, and q is *mixed* otherwise. $M = (Q, q_0, \delta)$ is *deterministic* if $\forall(q, \ell, q'), (q, \ell', q'') \in \delta : \ell = \ell' \implies q' = q''$. $M = (Q, q_0, \delta)$ is *send* (resp. *receive*) *directed* if for all sending (resp. receiving) $q \in Q$ and $(q, \ell, q'), (q, \ell', q'') \in \delta : \text{obj}(\ell) = \text{obj}(\ell')$. M is *directed* if it is send and receive directed.

Remark 1. In this paper, we consider only deterministic communicating automata without mixed states, and call them *Communicating Session Automata* (CSA). We discuss possible extensions of our results beyond this class in Sect. 7.

Definition 2 (System). Given a communicating automaton $M_p = (Q_p, q_{0p}, \delta_p)$ for each $p \in \mathcal{P}$, the tuple $S = (M_p)_{p \in \mathcal{P}}$ is a system. A configuration of S is a pair $s = (\mathbf{q}; \mathbf{w})$ where $\mathbf{q} = (q_p)_{p \in \mathcal{P}}$ with $q_p \in Q_p$ and where $\mathbf{w} = (w_{pq})_{pq \in \mathcal{C}}$ with $w_{pq} \in \Sigma^*$; component \mathbf{q} is the control state and $q_p \in Q_p$ is the local state of automaton M_p . The initial configuration of S is $s_0 = (\mathbf{q}_0; \epsilon)$ where $\mathbf{q}_0 = (q_{0p})_{p \in \mathcal{P}}$ and we write ϵ for the $|\mathcal{C}|$ -tuple $(\epsilon, \dots, \epsilon)$.

Hereafter, we fix a communicating session automaton $M_p = (Q_p, q_{0p}, \delta_p)$ for each $p \in \mathcal{P}$ and let $S = (M_p)_{p \in \mathcal{P}}$ be the corresponding system whose initial configuration is s_0 . For each $p \in \mathcal{P}$, we assume that $\forall(q, \ell, q') \in \delta_p : \text{subj}(\ell) = p$. We assume that the components of a configuration are named consistently, e.g., for $s' = (\mathbf{q}'; \mathbf{w}')$, we implicitly assume that $\mathbf{q}' = (q'_p)_{p \in \mathcal{P}}$ and $\mathbf{w}' = (w'_{pq})_{pq \in \mathcal{C}}$.

Definition 3 (Reachable configuration). Configuration $s' = (\mathbf{q}'; \mathbf{w}')$ is reachable from configuration $s = (\mathbf{q}; \mathbf{w})$ by firing transition ℓ , written $s \xrightarrow{\ell} s'$ (or $s \rightarrow s'$ when ℓ is not relevant), if there are $s, r \in \mathcal{P}$ and $a \in \Sigma$ such that either:

1. (a) $\ell = \mathbf{sr}!a$ and $(q_s, \ell, q'_s) \in \delta_s$, (b) $q'_p = q_p$ for all $p \neq s$, (c) $w'_{\mathbf{sr}} = w_{\mathbf{sr}} \cdot a$ and $w'_{pq} = w_{pq}$ for all $pq \neq \mathbf{sr}$; or
2. (a) $\ell = \mathbf{sr}?a$ and $(q_r, \ell, q'_r) \in \delta_r$, (b) $q'_p = q_p$ for all $p \neq r$, (c) $w_{\mathbf{sr}} = a \cdot w'_{\mathbf{sr}}$, and $w'_{pq} = w_{pq}$ for all $pq \neq \mathbf{sr}$.

Remark 2. Hereafter, we assume that any bound k is finite and $k \in \mathbb{N}_{>0}$.

We write \rightarrow^* for the reflexive and transitive closure of \rightarrow . Configuration $(\mathbf{q}; \mathbf{w})$ is k -bounded if $\forall pq \in \mathcal{C} : |w_{pq}| \leq k$. We write $s_1 \xrightarrow{\ell_1 \dots \ell_n} s_{n+1}$ when $s_1 \xrightarrow{\ell_1} s_2 \dots s_n \xrightarrow{\ell_n} s_{n+1}$, for some s_2, \dots, s_n (with $n \geq 0$); and say that the execution $\ell_1 \dots \ell_n$ is k -bounded from s_1 if $\forall 1 \leq i \leq n+1 : s_i$ is k -bounded. Given $\phi \in \mathcal{A}^*$, we write $p \notin \phi$ iff $\phi = \phi_0 \cdot \ell \cdot \phi_1 \implies \text{subj}(\ell) \neq p$. We write $s \xrightarrow{\phi}_k s'$ if s' is reachable with a k -bounded execution ϕ from s . The set of *reachable configurations of S* is $RS(S) = \{s \mid s_0 \rightarrow^* s\}$. The k -reachability set of S is the largest subset $RS_k(S)$ of $RS(S)$ within which each configuration s can be reached by a k -bounded execution from s_0 .

Definition 4 streamlines notions of safety from previous works [6, 12, 18, 39] (absence of deadlocks, orphan messages, and unspecified receptions).

Definition 4 (k -Safety). S is k -safe if the following holds $\forall (\mathbf{q}; \mathbf{w}) \in RS_k(S)$:

(ER) $\forall pq \in \mathcal{C}$, if $w_{pq} = a \cdot w'$, then $(\mathbf{q}; \mathbf{w}) \xrightarrow{k^*} \xrightarrow{pq?a} k$.

(PG) $\forall p \in \mathcal{P}$, if q_p is receiving, then $(\mathbf{q}; \mathbf{w}) \xrightarrow{k^*} \xrightarrow{qp?a} k$ for $q \in \mathcal{P}$ and $a \in \Sigma$.

We say that S is safe if it validates the unbounded version of k -safety (∞ -safe).

Property (ER), called *eventual reception*, requires that any sent message can always eventually be received (i.e., if a is the head of a queue then there must be an execution that consumes a), and Property (PG), called *progress*, requires that any automaton in a receiving state can eventually make a move (i.e., it can always eventually receive an *expected message*).

We say that a configuration s is *stable* iff $s = (\mathbf{q}; \epsilon)$, i.e., all its queues are empty. Next, we define the *stable property* for systems of communicating automata, following the definition from [18].

Definition 5 (Stable). S has the stable property (SP) if $\forall s \in RS(S) : \exists (\mathbf{q}; \epsilon) \in RS(S) : s \rightarrow^*(\mathbf{q}; \epsilon)$.

A system has the stable property if it is possible to reach a stable configuration from any reachable configuration. This property is called *deadlock-free* in [22]. The stable property implies the eventual reception property, but not safety (e.g., an automaton may be waiting for an input in a stable configuration, see Example 2), and safety does not imply the stable property, see Example 4.

Example 2. The following system has the stable property, but it is not safe.

$$M_s : \xleftarrow{\text{pq!}a} \circlearrowleft \xrightarrow{\text{pq!}b} \circlearrowright \quad M_q : \xleftarrow{\text{pq?}a} \circlearrowleft \xrightarrow{\text{pq?}b} \circlearrowright \quad M_r : \xleftarrow{\text{qr?}c} \circlearrowleft \xrightarrow{\text{qr?}c} \circlearrowright$$

Next, we define two properties related to *bound independence*. They specify classes of CSA whose branching behaviours are not affected by channel bounds.

Definition 6 (k -OBI). S is k -output bound independent (k -OBI), if $\forall s = (\mathbf{q}; \mathbf{w}) \in RS_k(S)$ and $\forall p \in \mathcal{P}$, if $s \xrightarrow{\text{pq}!a} k$, then $\forall (q_p, \text{pr}!b, q'_p) \in \delta_p : s \xrightarrow{\text{pr}!b} k$.

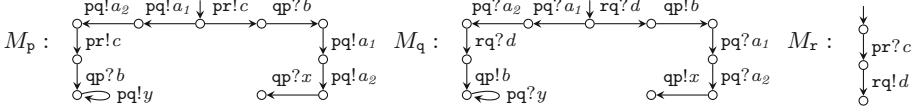


Fig. 2. Example of a *non-OBI* and *non-safe* system.

Definition 7 (k -IBI). S is k -input bound independent (k -IBI), if $\forall s = (\mathbf{q}; \mathbf{w}) \in RS_k(S)$ and $\forall p \in \mathcal{P}$, if $s \xrightarrow{\text{qp}?a} k$, then $\forall \ell \in \mathcal{A} : s \xrightarrow{\ell} k \wedge \text{subj}(\ell) = p \Rightarrow \ell = \text{qp}?a$.

If S is k -OBI, then any automaton that reaches a sending state is able to fire any of its available transitions, i.e., sending states model *internal choices* which are not constrained by bounds greater than or equal to k . Note that the unbounded version of k -OBI ($k = \infty$) is trivially satisfied for any system due to unbounded asynchrony. If S is k -IBI, then any automaton that reaches a receiving state is able to fire at most one transition, i.e., receiving states model *external choices* where the behaviour of the receiving automaton is controlled exclusively by its environment. We write IBI for the unbounded version of k -IBI ($k = \infty$).

Checking the IBI property is generally undecidable. However, systems consisting of (send and receive) *directed* automata are trivially k -IBI and k -OBI for all k , this subclass of CSA was referred to as *basic* in [18]. We introduce larger decidable approximations of IBI with Definitions 10 and 11.

Proposition 1. (1) If S is send directed, then S is k -OBI for all $k \in \mathbb{N}_{>0}$. (2) If S is receive directed, then S is IBI (and k -IBI for all $k \in \mathbb{N}_{>0}$).

Remark 3. CSA validating k -OBI and IBI strictly include the most common forms of asynchronous multiparty session types, e.g., the directed CSA of [18], and systems obtained by projecting Scribble specifications (global types) which need to be receive directed (this is called “consistent external choice subjects” in [31]) and which validate 1-OBI by construction since they are projections of synchronous specifications where choices must be located at a unique sender.

3 Bounded Compatibility for CSA

In this section, we introduce k -multiparty compatibility (k -MC) and study its properties wrt. Safety of communicating session automata (CSA) which are k -OBI and IBI. Then, we soundly and completely characterise k -exhaustivity in terms of local bound-agnosticity, a property which guarantees that communicating automata behave equivalently under any bound greater than or equal to k .

3.1 Multiparty Compatibility

The definition of k -MC is divided in two parts: (i) k -*exhaustivity* guarantees that the set of k -reachable configurations contains enough information to make a sound decision wrt. safety of the system; and (ii) k -*safety* (Definition 4) guarantees that a subset of all possible executions is free of any communication errors. Next, we define k -exhaustivity, then k -multiparty compatibility. Intuitively, a system is k -exhaustive if for all k -reachable configurations, whenever a send action is enabled, then it can be fired within a k -bounded execution.

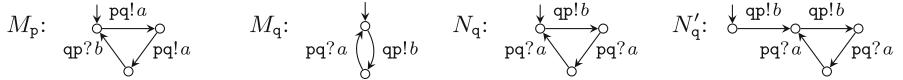


Fig. 3. (M_p, M_q) is non-exhaustive, (M_p, N_q) is 1-exhaustive, (M_p, N'_q) is 2-exhaustive.

Definition 8 (k -Exhaustivity). S is k -exhaustive if $\forall(\mathbf{q}; \mathbf{w}) \in RS_k(S)$ and $\forall p \in \mathcal{P}$, if q_p is sending, then $\forall(q_p, \ell, q'_p) \in \delta_p : \exists \phi \in \mathcal{A}^* : (\mathbf{q}; \mathbf{w}) \xrightarrow{\phi} \xrightarrow{k} \ell \wedge p \notin \phi$.

Definition 9 (k -Multiparty compatibility). S is k -multiparty compatible (k -MC) if it is k -safe and k -exhaustive.

Definition 9 is a natural extension of the definitions of *synchronous* multiparty compatibility given in [18, Definition 4.2] and [6, Definition 4]. The common key requirements are that *every send action must be matched by a receive action* (i.e., send actions are universally quantified), while *at least one receive action must find a matching send action* (i.e., receive actions are existentially quantified). Here, the universal check on send actions is done via the eventual reception property (ER) and the k -exhaustivity condition; while the existential check on receive actions is dealt with by the progress property (PG).

Whenever systems are k -OBI and IBI, then k -exhaustivity implies that k -bounded executions are sufficient to make a sound decision wrt. safety. This is not necessarily the case for systems outside of this class, see Examples 3 and 5.

Example 3. The system (M_p, M_q, M_r) in Fig. 2 is k -OBI for any k , but not IBI (it is 1-IBI but not k -IBI for any $k \geq 2$). When executing with a bound strictly greater than 1, there is a configuration where M_q is in its initial state and *both* its receive transitions are enabled. The system is 1-safe and 1-exhaustive (hence 1-MC) but it is *not* 2-exhaustive nor 2-safe. By constraining the automata to execute with a channel bound of 1, the left branch of M_p is prevented to execute together with the right branch of M_q . Thus, the fact that the y messages are not received in this case remains invisible in 1-bounded executions. This example can be easily extended so that it is n -exhaustive (resp. safe) but not $n+1$ -exhaustive (resp. safe) by sending/receiving $n+1$ a_i messages.

Example 4. The system in Fig. 1 is *directed* and 1-MC. The system (M_p, M_q) in Fig. 3 is safe but *not k-MC* for any finite $k \in \mathbb{N}_{>0}$. Indeed, for any execution of this system, at least one of the queues grows arbitrarily large. The system (M_p, N_q) is 1-MC while the system (M_p, N'_q) is *not* 1-MC but it is 2-MC.

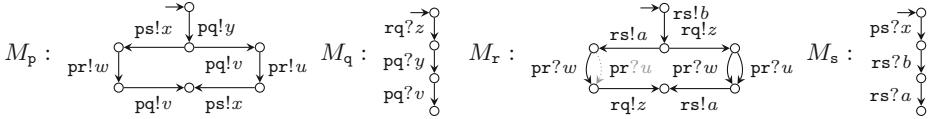


Fig. 4. Example of a system which is not 1-OBI.

Example 5. The system in Fig. 4 (without the dotted transition) is 1-MC, but not 2-safe; it is not 1-OBI but it is 2-OBI. In 1-bounded executions, M_r can execute $rs!b \cdot rp!z$, but it cannot fire $rs!b \cdot rs!a$ (queue rs is full), which violates the 1-OBI property. The system with the dotted transition is not 1-OBI, but it is 2-OBI and k -MC for any $k \geq 1$. Both systems are receive directed, hence IBI.

Theorem 1. *If S is k -OBI, IBI, and k -MC, then it is safe.*

Remark 4. It is undecidable whether there exists a bound k for which an arbitrary system is k -MC. This is a consequence of the Turing completeness of communicating (session) automata [10, 20, 42].

Although the IBI property is generally undecidable, it is possible to identify sound approximations, as we show below. We adapt the dependency relation from [39] and say that action ℓ' depends on ℓ from $s = (\mathbf{q}; \mathbf{w})$, written $s \vdash \ell < \ell'$, iff $subj(\ell) = subj(\ell') \vee (chan(\ell) = chan(\ell') \wedge w_{chan(\ell)} = \epsilon)$. Action ℓ' depends on ℓ in ϕ from s , written $s \vdash \ell <_\phi \ell'$, if the following holds:

$$s \vdash \ell <_\phi \ell' \iff \begin{cases} (s \vdash \ell < \ell'' \wedge s \vdash \ell'' <_\psi \ell') \vee s \vdash \ell <_\psi \ell' & \text{if } \phi = \ell'' \cdot \psi \\ s \vdash \ell < \ell' & \text{otherwise} \end{cases}$$

Definition 10. S is k -chained input bound independent (k -CIBI) if $\forall s = (\mathbf{q}; \mathbf{w}) \in RS_k(S)$ and $\forall p \in \mathcal{P}$, if $s \xrightarrow{\mathbf{q}p?a}_k s'$, then $\forall (q_p, sp?b, q'_p) \in \delta_p : s \neq q \implies \neg(s \xrightarrow{sp?b}_k) \wedge (\forall \phi \in \mathcal{A}^* : s' \xrightarrow{\phi} s \xrightarrow{sp!b}_k \implies s \vdash qp?a <_\phi sp!b)$.

Definition 11. S is k -strong input bound independent (k -SIBI) if $\forall s = (\mathbf{q}; \mathbf{w}) \in RS_k(S)$ and $\forall p \in \mathcal{P}$, if $s \xrightarrow{\mathbf{q}p?a}_k s'$, then $\forall (q_p, sp?b, q'_p) \in \delta_p : s \neq q \implies \neg(s \xrightarrow{sp?b}_k \vee s' \xrightarrow{* sp!b}_k)$.

Definition 10 requires that whenever p can fire a receive action, at most one of its receive actions is enabled at s , and no other receive transition from q_p will be enabled until p has made a move. This is due to the existence of a dependency chain between the reception of a message $(qp?a)$ and the matching send of another possible reception $(sp!b)$. Property k -SIBI (Definition 11) is a stronger version of k -CIBI, which can be checked more efficiently.

Lemma 1. *If S is k -OBI, k -CIBI (resp. k -SIBI) and k -exhaustive, then it is IBI.*

The decidability of k -OBI, k -IBI, k -SIBI, k -CIBI, and k -MC is straightforward since both $RS_k(S)$ (which has an exponential number of states wrt. k) and \rightarrow_k are finite, given a finite k . Theorem 2 states the space complexity of the procedures, except for k -CIBI for which a complexity class is yet to be determined. We show that the properties are PSPACE by reducing to an instance of the reachability problem over a transition system built following the construction of Bollig et al. [8, Theorem 6.3]. The rest of the proof follows from similar arguments in Genest et al. [22, Proposition 5.5] and Bouajjani et al. [9, Theorem 3].

Theorem 2. *The problems of checking the k -OBI, k -IBI, k -SIBI, k -safety, and k -exhaustivity properties are all decidable and PSPACE-complete (with $k \in \mathbb{N}_{>0}$ given in unary). The problem of checking the k -CIBI property is decidable.*

3.2 Local Bound-Agnosticity

We introduce local bound-agnosticity and show that it fully characterises k -exhaustive systems. Local bound-agnosticity guarantees that each communicating automaton behave in the same manner for any bound greater than or equal to some k . Therefore such systems may be executed transparently under a bounded semantics (a communication model available in Go and Rust).

Definition 12 (Transition system). *The k -bounded transition system of S is the labelled transition system (LTS) $TS_k(S) = (N, s_0, \Delta)$ such that $N = RS_k(S)$, s_0 is the initial configuration of S , $\Delta \subseteq N \times \mathcal{A} \times N$ is the transition relation, and $(s, \ell, s') \in \Delta$ if and only if $s \xrightarrow{k}^\ell s'$.*

Definition 13 (Projection). *Let \mathcal{T} be an LTS over \mathcal{A} . The projection of \mathcal{T} onto p , written $\pi_p^\epsilon(\mathcal{T})$, is obtained by replacing each label ℓ in \mathcal{T} by $\pi_p(\ell)$.*

Recall that the projection of action ℓ , written $\pi_p(\ell)$, is defined in Sect. 2. The automaton $\pi_p^\epsilon(TS_k(S))$ is essentially the *local* behaviour of participant p within the transition system $TS_k(S)$. When each automaton in a system S behaves equivalently for any bound greater than or equal to some k , we say that S is locally bound-agnostic. Formally, S is *locally bound-agnostic for k* when $\pi_p^\epsilon(TS_k(S))$ and $\pi_p^\epsilon(TS_n(S))$ are weakly bisimilar (\approx) for each participant p and any $n \geq k$. For k -OBI and IBI systems, local bound-agnosticity is a *necessary and sufficient* condition for k -exhaustivity, as stated in Theorem 3 and Corollary 1.

Theorem 3. Let S be a system.

- (1) If $\exists k \in \mathbb{N}_{>0} : \forall p \in \mathcal{P} : \pi_p^\epsilon(TS_k(S)) \approx \pi_p^\epsilon(TS_{k+1}(S))$, then S is k -exhaustive.
- (2) If S is k -OBI, IBI, and k -exhaustive, then $\forall p \in \mathcal{P} : \pi_p^\epsilon(TS_k(S)) \approx \pi_p^\epsilon(TS_{k+1}(S))$.

Corollary 1. Let S be k -OBI and IBI s.t. $\forall p \in \mathcal{P} : \pi_p^\epsilon(TS_k(S)) \approx \pi_p^\epsilon(TS_{k+1}(S))$, then S is locally bound-agnostic for k .

Theorem 3 (1) is reminiscent of the (PSPACE-complete) checking procedure for existentially bounded systems with the stable property [22] (an *undecidable* property). Recall that k -exhaustivity is not sufficient to guarantee safety, see Examples 3 and 5. We give an effective procedure (based on partial order reduction) to check k -exhaustivity and related properties in [43].

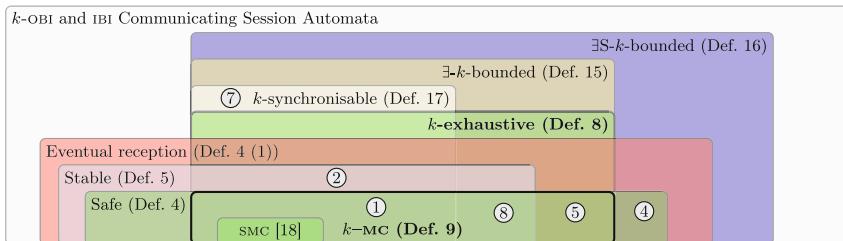


Fig. 5. Relations between k -exhaustivity, existential k -boundedness, and k -synchronisability in k -OBI and IBI CSA (the circled numbers refer to Table 1).

4 Existentially Bounded and Synchronisable Automata

4.1 Kuske and Muscholl's Existential Boundedness

Existentially bounded communicating automata [21, 22, 35] are a class of communicating automata whose executions can always be scheduled in such a way that the number of pending messages is bounded by a given value. Traditionally, existentially bounded communicating automata are defined on communicating automata that feature (local) accepting states and in terms of *accepting runs*. An accepting run is an execution (starting from s_0) which terminates in a configuration $(\mathbf{q}; \mathbf{w})$ where each q_p is a local accepting state. In our setting, we simply consider that every local state q_p is an accepting state, hence any execution ϕ starting from s_0 is an accepting run. We first study existential boundedness as defined in [35] as it matches more closely k -exhaustivity, we study the “classical” definition of existential boundedness [22] in Sect. 4.2.

Following [35], we say that an execution $\phi \in \mathcal{A}^*$ is *valid* if for any prefix ψ of ϕ and any channel $pq \in \mathcal{C}$, we have that $\pi_{pq}^?(\psi)$ is a prefix of $\pi_{pq}^!(\psi)$, i.e., an execution is valid if it models the FIFO semantics of communicating automata.

Definition 14 (Causal equivalence [35]). Given $\phi, \psi \in \mathcal{A}^*$, we define: $\phi \approx \psi$ iff ϕ and ψ are valid executions and $\forall p \in \mathcal{P} : \pi_p(\phi) = \pi_p(\psi)$. We write $[\phi]_\approx$ for the equivalence class of ϕ wrt. \approx .

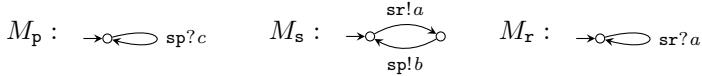
Definition 15 (Existential boundedness [35]). We say that a valid execution ϕ is k -match-bounded if, for every prefix ψ of ϕ the difference between the number of matched events of type $pq!$ and those of type $pq?$ is bounded by k , i.e., $\min\{|\pi_{pq}^!(\psi)|, |\pi_{pq}^?(\phi)|\} - |\pi_{pq}^?(\psi)| \leq k$.

Write $\mathcal{A}^*|_k$ for the set of k -match-bounded words. An execution ϕ is existentially k -bounded if $[\phi]_\approx \cap \mathcal{A}^*|_k \neq \emptyset$. A system S is existentially k -bounded, written \exists - k -bounded, if each execution in $\{\phi \mid \exists s : s_0 \xrightarrow{\phi} s\}$ is existentially k -bounded.

Example 6. Consider Fig. 3. (M_p, M_q) is not existentially k -bounded, for any k : at least one of the queues must grow infinitely for the system to progress. Systems (M_p, N_q) and (M_p, N'_q) are existentially bounded since any of their executions can be scheduled to an \approx -equivalent execution which is 2-match-bounded.

The relationship between k -exhaustivity and existential boundedness is stated in Theorem 4 and illustrated in Fig. 5 for k -OBI and IBI CSA, where SMC refers to synchronous multiparty compatibility [18, Definition 4.2]. The circled numbers in the figure refer to key examples summarised in Table 1. The strict inclusion of k -exhaustivity in existential k -boundedness is due to systems that do not have the eventual reception property, see Example 7.

Example 7. The system below is \exists -1-bounded but is not k -exhaustive for any k .



For any k , the channel sp eventually gets full and the send action $sp!b$ can no longer be fired; hence it does not satisfy k -exhaustivity. Note that each execution can be reordered into a 1-match-bounded execution (the b 's are never matched).

Theorem 4. (1) If S is k -OBI, IBI, and k -exhaustive, then it is \exists - k -bounded.
(2) If S is \exists - k -bounded and satisfies eventual reception, then it is k -exhaustive.

4.2 Existentially Stable Bounded Communicating Automata

The “classical” definition of existentially bounded communicating automata as found in [22] differs slightly from Definition 15, as it relies on a different notion of accepting runs, see [22, page 4]. Assuming that all local states are accepting, we adapt their definition as follows: a *stable accepting run* is an execution ϕ starting from s_0 which terminates in a *stable configuration*.

Definition 16 (Existential stable boundedness [22]). A system S is existentially stable k -bounded, written $\exists S$ - k -bounded, if for each execution ϕ in $\{\phi \mid \exists (q; \epsilon) \in RS(S) : s_0 \xrightarrow{\phi} (q; \epsilon)\}$ there is ψ such that $s_0 \xrightarrow{\psi} k$ with $\phi \approx \psi$.

A system is existentially stable k -bounded if each of its executions leading to a *stable* configuration can be re-ordered into a k -bounded execution (from s_0).

Theorem 5. (1) If S is existentially k -bounded, then it is existentially stable k -bounded. (2) If S is existentially stable k -bounded and has the stable property, then it is existentially k -bounded.

We illustrate the relationship between existentially stable bounded communicating automata and the other classes in Fig. 5. The example below further illustrates the strictness of the inclusions, see Table 1 for a summary.

Example 8. Consider the systems in Fig. 3. (M_p, M_q) and (M_p, N'_q) are (trivially) existentially stable 1-bounded since none of their (non-empty) executions terminate in a stable configuration. The system (M_p, N_q) is existentially stable 2-bounded since each of its executions can be re-ordered into a 2-bounded one. The system in Example 7 is (trivially) $\exists S$ -1-bounded: none of its (non-empty) executions terminate in a stable configuration (the b 's are never received).

Theorem 6. Let S be an $\exists(S)$ - k -bounded system with the stable property, then it is k -exhaustive.

Table 1. Properties for key examples, where direct. stands for directed, OBI for k -OBI, SIBI for k -SIBI, ER for eventual reception property, SP for stable property, exh. for k -exhaustive, $\exists(S)$ -b for $\exists(S)$ -bounded, and syn. for n -synchronisable (for some $n \in \mathbb{N}_{>0}$).

#	System	Ref.	k	direct.	OBI	SIBI	safe	ER	SP	exh.	$\exists S$ -b	\exists -b	syn.
1	(M_c, M_s, M_1)	Figure 1	1	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes
2	(M_s, M_q, M_r)	Example 2	1	yes	yes	yes	no	yes	yes	yes	yes	yes	yes
3	(M_p, M_q, M_r)	Figure 2	≥ 2	no	yes	no	no	no	no	no	yes	yes	no
4	(M_p, M_q)	Figure 3	any	yes	yes	yes	yes	yes	no	no	yes	no	no
5	(M_p, N'_q)	Figure 3	2	yes	yes	yes	yes	yes	no	yes	yes	yes	no
6	(M_p, M_q, M_r, M_s)	Figure 4	2	no	yes	yes	yes	yes	no	yes	yes	yes	no
7	(M_s, M_r, M_p)	Example 7	any	yes	yes	yes	no	no	no	no	yes	yes	yes
8	(M_p, M_q)	Example 9	1	yes	yes	yes	yes	yes	yes	yes	yes	yes	no

4.3 Synchronisable Communicating Session Automata

In this section, we study the relationship between synchronisability [9] and k -exhaustivity via existential boundedness. Informally, communicating automata are synchronisable if each of their executions can be scheduled in such a way that it consists of sequences of “exchange phases”, where each phase consists of a bounded number of send actions, followed by a sequence of receive actions. The original definition of k -synchronisable systems [9, Definition 1] is based on

communicating automata with *mailbox* semantics, i.e., each automaton has one input queue. Here, we adapt the definition so that it matches our point-to-point semantics. We write $\mathcal{A}_!$ for $\mathcal{A} \cap (\mathcal{C} \times \{!\} \times \Sigma)$, and $\mathcal{A}_?$ for $\mathcal{A} \cap (\mathcal{C} \times \{?\} \times \Sigma)$.

Definition 17 (Synchronisability). A valid execution $\phi = \phi_1 \cdots \phi_n$ is a k -exchange if and only if: (1) $\forall 1 \leq i \leq n : \phi_i \in \mathcal{A}_!^* \cdot \mathcal{A}_?^* \wedge |\phi_i| \leq 2k$; and
(2) $\forall p, q \in \mathcal{C} : \forall 1 \leq i \leq n : \pi_{pq}^!(\phi_i) \neq \pi_{pq}^?(\phi_i) \implies \forall i < j \leq n : \pi_{pq}^?(\phi_j) = \epsilon$.

We write $\mathcal{A}^* \|_k$ for the set of executions that are k -exchanges and say that an execution ϕ is k -synchronisable if $[\phi]_\diamond \cap \mathcal{A}^* \|_k \neq \emptyset$. A system S is k -synchronisable if each execution in $\{\phi \mid \exists s : s_0 \xrightarrow{\phi} s\}$ is k -synchronisable.

Table 2. Experimental evaluation. $|\mathcal{P}|$ is the number of participants, k is the bound, $|RTS|$ is the number of transitions in the reduced $TS_k(S)$ (see [43]), direct. stands for directed, Time is the time taken to check all the properties shown in this table, and GMC is yes if the system is generalised multiparty compatible [39].

Example	$ \mathcal{P} $	k	$ RTS $	direct.	$k\text{-OBI}$	$k\text{-CIBI}$	$k\text{-MC}$	Time	GMC
Client-Server-Logger	3	1	11	yes	yes	yes	yes	0.04 s	no
4 Player game [†] [39]	4	1	20	no	yes	yes	yes	0.05 s	yes
Bargain [39]	3	1	8	yes	yes	yes	yes	0.03 s	yes
Filter collaboration [68]	2	1	10	yes	yes	yes	yes	0.03 s	yes
Alternating bit [†] [59]	2	1	8	yes	yes	yes	yes	0.04 s	no
TPMContract v2 [†] [25]	2	1	14	yes	yes	yes	yes	0.04 s	yes
Sanitary agency [†] [60]	4	1	34	yes	yes	yes	yes	0.07 s	yes
Logistic [†] [54]	4	1	26	yes	yes	yes	yes	0.05 s	yes
Cloud system v4 [24]	4	2	16	no	yes	yes	yes	0.04 s	yes
Commit protocol [9]	4	1	12	yes	yes	yes	yes	0.03 s	yes
Elevator [†] [9]	5	1	72	no	yes	no	yes	0.14 s	no
Elevator-dashed [†] [9]	5	1	80	no	yes	no	yes	0.16 s	no
Elevator-directed [†] [9]	3	1	41	yes	yes	yes	yes	0.07 s	yes
Dev system [58]	4	1	20	yes	yes	yes	yes	0.05 s	no
Fibonacci [48]	2	1	6	yes	yes	yes	yes	0.03 s	yes
SAP-Negot. [48, 53]	2	1	18	yes	yes	yes	yes	0.04 s	yes
SH [48]	3	1	30	yes	yes	yes	yes	0.06 s	yes
Travel agency [48, 64]	3	1	21	yes	yes	yes	yes	0.05 s	yes
HTTP [29, 48]	2	1	48	yes	yes	yes	yes	0.07 s	yes
SMTP [30, 48]	2	1	108	yes	yes	yes	yes	0.08 s	yes
gen_server (buggy) [67]	3	1	56	no	no	yes	no	0.03 s	no
gen_server (fixed) [67]	3	1	45	no	yes	yes	yes	0.03 s	yes
Double buffering [45]	3	2	16	yes	yes	yes	yes	0.01 s	no

Condition (1) says that execution ϕ should be a sequence of an arbitrary number of send-receive phases, where each phase consists of at most $2k$ actions. Condition (2) says that if a message is not received in the phase in which it is sent, then it cannot be received in ϕ . Observe that the bound k is on the number of actions (over possibly different channels) in a phase rather than the number of pending messages in a given channel.

Example 9. The system below (left) is 1-MC and $\exists(S)$ -1-bounded, but it is *not* k -synchronisable for any k . The subsequences of send-receive actions in the \approx -equivalent executions below (right).

$$\begin{array}{c|c} M_p : \quad \xrightarrow{\text{pq! } a} \xrightarrow{\text{qp? } c} \xrightarrow{\text{pq! } b} \xrightarrow{\text{qp? } d} \\ \xrightarrow{\text{---}} \xrightarrow{\text{---}} \xrightarrow{\text{---}} \xrightarrow{\text{---}} \\ M_q : \quad \xrightarrow{\text{qp! } c} \xrightarrow{\text{qp! } d} \xrightarrow{\text{pq? } a} \xrightarrow{\text{pq? } b} \\ \xrightarrow{\text{---}} \xrightarrow{\text{---}} \xrightarrow{\text{---}} \xrightarrow{\text{---}} \end{array} \quad \begin{array}{l} \phi_1 = \boxed{\text{pq! } a \cdot \text{qp! } c \cdot \text{qp? } c \cdot \text{qp! } d} \cdot \boxed{\text{pq? } a \cdot \text{pq! } b \cdot \text{qp? } d} \cdot \boxed{\text{pq? } b} \\ \phi_2 = \boxed{\text{pq! } a \cdot \text{qp! } c \cdot \text{qp! } d} \cdot \boxed{\text{qp? } c \cdot \text{pq? } a} \cdot \boxed{\text{pq! } b \cdot \text{qp? } d} \cdot \boxed{\text{pq? } b} \end{array}$$

Execution ϕ_1 is 1-bounded for s_0 , but it is not a k -exchange since, e.g., a is received outside of the phase where it is sent. In ϕ_2 , message d is received outside of its sending phase. In the terminology of [9], this system is not k -synchronisable because there is a “receive-send dependency” between the exchange of message c and b , i.e., p must receive c before it sends b . Hence, there is no k -exchange that is \approx -equivalent to ϕ_1 and ϕ_2 .

Theorem 7. (1) If S is k -synchronisable, then it is \exists - k -bounded. (2) If S is k -synchronisable and has the eventual reception property, then it is k -exhaustive.

Figure 5 and Table 1 summarise the results of Sect. 4 wrt. k -OBI and IBI CSA. We note that any finite-state system is k -exhaustive (and $\exists(S)$ - k -bounded) for sufficiently large k , while this does not hold for synchronisability, see Example 9.

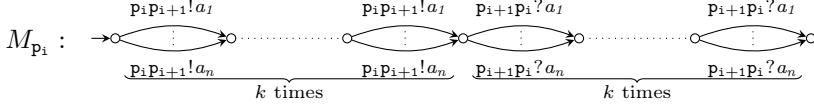
5 Experimental Evaluation

We have implemented our theory in a tool [33] which takes two inputs: (i) a system of communicating automata and (ii) a bound MAX. The tool iteratively checks whether the system validates the premises of Theorem 1, until it succeeds or reaches $k = \text{MAX}$. We note that the k -OBI and IBI conditions are required for our soundness result (Theorem 1), but are orthogonal for checking k -MC. Each condition is checked on a *reduced bounded transition system*, called $RTS_k(S)$. Each verification procedure for these conditions is implemented in Haskell using a simple (depth-first-search based) reachability check on the paths of $RTS_k(S)$. We give an (optimal) partial order reduction algorithm to construct $RTS_k(S)$ in [43] and show that it preserves our properties.

We have tested our tool on 20 examples taken from the literature, which are reported in Table 2. The table shows that the tool terminates virtually instantaneously on all examples. The table suggests that many systems are indeed k -MC and most can be easily adapted to validate bound independence. The last column refers to the GMC condition, a form of *synchronous* multiparty compatibility (SMC) introduced in [39]. The examples marked with \dagger have been slightly

modified to make them CSA that validate k -OBI and IBI. For instance, we take only one of the possible interleavings between mixed actions to remove mixed states (taking send action before receive action to preserve safety), see [43].

We have assessed the scalability of our approach with automatically generated examples, which we report in Fig. 6. Each system considered in these benchmarks consists of $2m$ (directed) CSA for some $m \geq 1$ such that $S = (M_{p_i})_{1 \leq i \leq 2m}$, and each automaton M_{p_i} is of the form (when i is odd):



Each M_{p_i} sends k messages to participant p_{i+1} , then receives k messages from p_{i+1} . Each message is taken from an alphabet $\{a_1, \dots, a_n\}$ ($n \geq 1$). M_{p_i} has the same structure when i is even, but interacts with p_{i-1} instead. Observe that any system constructed in this way is k -MC for any $k \geq 1$, $n \geq 1$, and $m \geq 1$. The shape of these systems allows us to assess how our approach fares in the worst case, i.e., large number of paths in $RTS_k(S)$. Figure 6 gives the time taken for our tool to terminate (y axis) wrt. the number of transitions in $RTS_k(S)$ where k is the least natural number for which the system is k -MC. The plot on the left in Fig. 6 gives the timings when k is increasing (every increment from $k = 2$ to $k = 100$) with the other parameters fixed ($n = 1$ and $m = 5$). The middle plot gives the timings when m is increasing (every increment from $m = 1$ to $m = 26$) with $k = 10$ and $n = 1$. The right-hand side plot gives the timings when n is increasing (every increment from $n = 1$ to $n = 10$) with $k = 2$ and $m = 1$. The largest $RTS_k(S)$ on which we have tested our tool has 12222 states and 22220 transitions, and the verification took under 17 min.¹ Observe that partial order reduction mitigates the increasing size of the transition system on which k -MC is checked, e.g., these experiments show that parameters k and m have only a linear effect on the number of transitions (see horizontal distances between data points). However the number of transitions increases exponentially with n (since the number of paths in each automaton increases exponentially with n).

6 Related Work

Theory of communicating automata Communicating automata were introduced, and shown to be Turing powerful, in the 1980s [10] and have since then been studied extensively, namely through their connection with message sequence charts (MSC) [46]. Several works achieved decidability results by using bag or lossy channels [1, 2, 13, 14] or by restricting the topology of the network [36, 57].

Existentially bounded communicating automata stand out because they preserve the FIFO semantics of communicating automata, do not restrict the topology of the network, and include infinite state systems. Given a bound k and

¹ All the benchmarks in this paper were run on an 8-core Intel i7-7700 machine with 16 GB RAM running a 64-bit Linux.

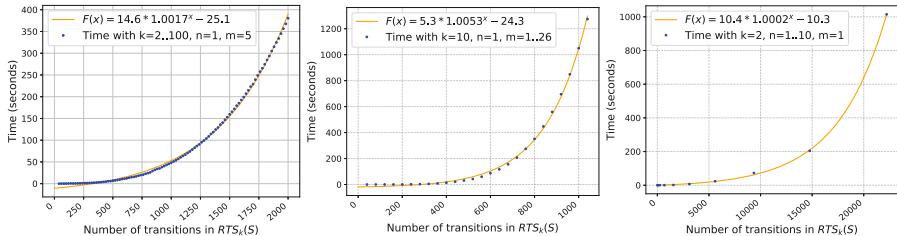


Fig. 6. Benchmarks: increasing k (left), increasing m (middle), and increasing n (right).

an arbitrary system of (deterministic) communicating automata S , it is generally *undecidable* whether S is existentially k -bounded. However, the question becomes decidable (PSPACE-complete) when S has the stable property. The stable property is itself generally *undecidable* (it is called deadlock-freedom in [22, 35]). Hence this class is *not* directly applicable to the verification of message passing programs since its membership is overall undecidable. We have shown that k -OBI, IBI, and k -exhaustive CSA systems are (strictly) included in the class of existentially bounded systems. Hence, our work gives a sound *practical* procedure to check whether CSA are existentially k -bounded. To the best of our knowledge, the only tools dedicated to the verification of (unbounded) communicating automata are McScM [26] and Chorgram [40]. Bouajjani et al. [9] study a variation of communicating automata with *mailboxes* (one input queue per automaton). They introduce the class of synchronisable systems and a procedure to check whether a system is k -synchronisable; it relies on executions consisting of k -bounded exchange phases. Given a system and a bound k , it is decidable (PSPACE-complete) whether its executions are equivalent to k -synchronous executions. Section 4.3 states that any k -synchronisable system which satisfies eventual reception is also k -exhaustive, see Theorem 7. In contrast to existential boundedness, synchronisability does not include all finite-state systems. Our characterisation result, based on local bound-agnosticity (Theorem 3), is *unique* to k -exhaustivity. It does not apply to existential boundedness nor synchronisability, see, e.g., Example 7. The term “synchronizability” is used by Basu et al. [3, 4] to refer to another verification procedure for communicating automata with mailboxes. Finkel and Lozes [19] have shown that this notion of synchronizability is undecidable. We note that a system that is safe with a point-to-point semantics, may not be safe with a mailbox semantics (due to independent send actions), and vice-versa. For instance, the system in Fig. 2 is safe when executed with mailbox semantics.

Multiparty Compatibility and Programming Languages. The first definition of multiparty compatibility appeared in [18, Definition 4.2], inspired by the work in [23], to characterise the relationship between global types and communicating automata. This definition was later adapted to the setting of communicating timed automata in [6]. Lange et al. [39] introduced a generalised version of multiparty compatibility (GMC) to support communicating automata that feature

mixed or non-directed states. Because our results apply to automata without mixed states, k -MC is not a strict extension of GMC, and GMC is not a strict extension of k -MC either, as it requires the existence of *synchronous* executions. In future work, we plan to develop an algorithm to synthesise representative choreographies from k -MC systems, using the algorithm in [39].

The notion of multiparty compatibility is at the core of recent works that apply session types techniques to programming languages. Multiparty compatibility is used in [51] to detect deadlocks in Go programs, and in [30] to study the well-formedness of Scribble protocols [64] through the compatibility of their projections. These protocols are used to generate various endpoint APIs that implement a Scribble specification [30, 31, 48], and to produce runtime monitoring tools [47, 49, 50]. Taylor et al. [67] use multiparty compatibility and choreography synthesis [39] to automate the analysis of the `gen_server` library of Erlang/OTP. We can transparently widen the set of safe programs captured by these tools by using k -MC instead of synchronous multiparty compatibility (SMC). The k -MC condition corresponds to a much wider instance of the *abstract* safety invariant φ for session types defined in [63]. Indeed k -MC includes SMC (see [43]) and all finite-state systems (for k sufficiently large).

7 Conclusions

We have studied CSA via a new condition called k -exhaustivity. The k -exhaustivity condition is (*i*) the basis for a wider notion of multiparty compatibility, k -MC, which captures asynchronous interactions and (*ii*) the first practical, empirically validated, sufficient condition for existential k -boundedness. We have shown that k -exhaustive systems are fully characterised by local bound-agnosticity (each automaton behaves equivalently for any bound greater than or equal to k). This is a key requirement for asynchronous message passing programming languages where the possibility of having infinitely many orphan messages is undesirable, in particular for Go and Rust which provide *bounded* communication channels.

For future work, we plan to extend our theory beyond CSA. We believe that it is possible to support mixed states and states which do not satisfy IBI, as long as their outgoing transitions are independent (i.e., if they commute). Additionally, to make k -MC checking more efficient, we will elaborate heuristics to find optimal bounds and off-load the verification of k -MC to an off-the-shelf model checker.

Acknowledgements. We thank Laura Bocchi and Alceste Scalas for their comments, and David Castro and Nicolas Dilley for testing the artifact. This work is partially supported by EPSRC EP/K034413/1, EP/K011715/1, EP/L00058X/1, EP/N027833/1, and EP/N028201/1.

References

1. Abdulla, P.A., Bouajjani, A., Jonsson, B.: On-the-fly analysis of systems with unbounded, lossy FIFO channels. In: Hu, A.J., Vardi, M.Y. (eds.) CAV 1998. LNCS, vol. 1427, pp. 305–318. Springer, Heidelberg (1998). <https://doi.org/10.1007/BFb0028754>
2. Abdulla, P.A., Jonsson, B.: Verifying programs with unreliable channels. In: LICS 1993, pp. 160–170 (1993)
3. Basu, S., Bultan, T.: Automated choreography repair. In: Stevens, P., Wąsowski, A. (eds.) FASE 2016. LNCS, vol. 9633, pp. 13–30. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49665-7_2
4. Basu, S., Bultan, T., Ouederni, M.: Deciding choreography realizability. In: POPL 2012, pp. 191–202 (2012)
5. Bocchi, L., Chen, T., Demangeon, R., Honda, K., Yoshida, N.: Monitoring networks through multiparty session types. *Theor. Comput. Sci.* **669**, 33–58 (2017)
6. Bocchi, L., Lange, J., Yoshida, N.: Meeting deadlines together. In: CONCUR 2015, pp. 283–296 (2015)
7. Bocchi, L., Yang, W., Yoshida, N.: Timed multiparty session types. In: Baldan, P., Gorla, D. (eds.) CONCUR 2014. LNCS, vol. 8704, pp. 419–434. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44584-6_29
8. Bollig, B., Kuske, D., Meinecke, I.: Propositional dynamic logic for message-passing systems. *Log. Methods Comput. Sci.* **6**(3) (2010). <https://lmcs.episciences.org/1057>
9. Bouajjani, A., Enea, C., Ji, K., Qadeer, S.: On the completeness of verifying message passing programs under bounded asynchrony. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10982, pp. 372–391. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96142-2_23
10. Brand, D., Zafropulo, P.: On communicating finite-state machines. *J. ACM* **30**(2), 323–342 (1983)
11. Castro, D., Hu, R., Jongmans, S., Ng, N., Yoshida, N.: Distributed programming using role-parametric session types in Go: statically-typed endpoint APIs for dynamically-instantiated communication structures. *PACMPL* **3**(POPL), 29:1–29:30 (2019)
12. Cécé, G., Finkel, A.: Verification of programs with half-duplex communication. *Inf. Comput.* **202**(2), 166–190 (2005)
13. Cécé, G., Finkel, A., Iyer, S.P.: Unreliable channels are easier to verify than perfect channels. *Inf. Comput.* **124**(1), 20–31 (1996)
14. Clemente, L., Herbreteau, F., Sutre, G.: Decidable topologies for communicating automata with FIFO and bag channels. In: Baldan, P., Gorla, D. (eds.) CONCUR 2014. LNCS, vol. 8704, pp. 281–296. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44584-6_20
15. Coppo, M., Dezani-Ciancaglini, M., Padovani, L., Yoshida, N.: A gentle introduction to multiparty asynchronous session types. In: Bernardo, M., Johnsen, E.B. (eds.) SFM 2015. LNCS, vol. 9104, pp. 146–178. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-18941-3_4
16. Demangeon, R., Honda, K., Hu, R., Neykova, R., Yoshida, N.: Practical interruptible conversations: distributed dynamic verification with multiparty session types and Python. *Form. Methods Syst. Des.* **46**(3), 197–225 (2015)
17. Deniélou, P.-M., Yoshida, N.: Multiparty session types meet communicating automata. In: Seidl, H. (ed.) ESOP 2012. LNCS, vol. 7211, pp. 194–213. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28869-2_10

18. Deniéou, P.-M., Yoshida, N.: Multiparty compatibility in communicating automata: characterisation and synthesis of global session types. In: Fomin, F.V., Freivalds, R., Kwiatkowska, M., Peleg, D. (eds.) ICALP 2013. LNCS, vol. 7966, pp. 174–186. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39212-2_18
19. Finkel, A., Lozes, É.: Synchronizability of communicating finite state machines is not decidable. In: ICALP 2017, pp. 122:1–122:14 (2017)
20. Finkel, A., McKenzie, P.: Verifying identical communicating processes is undecidable. *Theor. Comput. Sci.* **174**(1–2), 217–230 (1997)
21. Genest, B., Kuske, D., Muscholl, A.: A Kleene theorem and model checking algorithms for existentially bounded communicating automata. *Inf. Comput.* **204**(6), 920–956 (2006)
22. Genest, B., Kuske, D., Muscholl, A.: On communicating automata with bounded channels. *Fundam. Inform.* **80**(1–3), 147–167 (2007)
23. Gouda, M.G., Manning, E.G., Yu, Y.: On the progress of communications between two finite state machines. *Inf. Control* **63**(3), 200–216 (1984)
24. Güdemann, M., Salaün, G., Ouederni, M.: Counterexample guided synthesis of monitors for realizability enforcement. In: Chakraborty, S., Mukund, M. (eds.) ATVA 2012. LNCS, pp. 238–253. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33386-6_20
25. Hallé, S., Bultan, T.: Realizability analysis for message-based interactions using shared-state projections. In: SIGSOFT 2010, pp. 27–36 (2010)
26. Heußner, A., Le Gall, T., Sutre, G.: McScM: a general framework for the verification of communicating machines. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 478–484. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28756-5_34
27. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Hankin, C. (ed.) ESOP 1998. LNCS, vol. 1381, pp. 122–138. Springer, Heidelberg (1998). <https://doi.org/10.1007/BFb0053567>
28. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: POPL 2008, pp. 273–284 (2008)
29. Hu, R.: Distributed programming using Java APIs generated from session types. In: Behavioural Types: From Theory to Tools. River Publishers, June 2017
30. Hu, R., Yoshida, N.: Hybrid session verification through endpoint API generation. In: Stevens, P., Wąsowski, A. (eds.) FASE 2016. LNCS, vol. 9633, pp. 401–418. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49665-7_24
31. Hu, R., Yoshida, N.: Explicit connection actions in multiparty session types. In: FASE 2017, pp. 116–133 (2017)
32. Jespersen, T.B.L., Munksgaard, P., Larsen, K.F.: Session types for Rust. In: WGP@ICFP 2015, pp. 13–22 (2015)
33. KMC tool (2019). <https://bitbucket.org/julien-lange/kmc-cav19>
34. Kouzapas, D., Dardha, O., Perera, R., Gay, S.J.: Typechecking protocols with Mungo and StMungo. In: PPDP 2016, pp. 146–159 (2016)
35. Kuske, D., Muscholl, A.: Communicating automata (2014). <http://eiche.theoinf.tu-ilmenau.de/kuske/Submitted/cfm-final.pdf>
36. La Torre, S., Madhusudan, P., Parlato, G.: Context-bounded analysis of concurrent queue systems. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 299–314. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_21

37. Lange, J., Ng, N., Toninho, B., Yoshida, N.: Fencing off Go: liveness and safety for channel-based programming. In: POPL 2017, pp. 748–761 (2017)
38. Lange, J., Ng, N., Toninho, B., Yoshida, N.: A static verification framework for message passing in Go using behavioural types. In: ICSE 2018. ACM (2018)
39. Lange, J., Tuosto, E., Yoshida, N.: From communicating machines to graphical choreographies. In: POPL 2015, pp. 221–232 (2015)
40. Lange, J., Tuosto, E., Yoshida, N.: A tool for choreography-based analysis of message-passing software. In: Behavioural Types: from Theory to Tools. River Publishers, June 2017
41. Lange, J., Yoshida, N.: Characteristic formulae for session types. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 833–850. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_52
42. Lange, J., Yoshida, N.: On the undecidability of asynchronous session subtyping. In: Esparza, J., Murawski, A.S. (eds.) FoSSaCS 2017. LNCS, vol. 10203, pp. 441–457. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54458-7_26
43. Lange, J., Yoshida, N.: Verifying asynchronous interactions via communicating session automata. CoRR, abs/1901.09606 (2019). <https://arxiv.org/abs/1901.09606>
44. Lindley, S., Morris, J.G.: Embedding session types in Haskell. In: Haskell 2016, pp. 133–145 (2016)
45. Mostrou, D., Yoshida, N., Honda, K.: Global principal typing in partially commutative asynchronous sessions. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 316–332. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00590-9_23
46. Muscholl, A.: Analysis of communicating automata. In: Dediū, A.-H., Fernau, H., Martín-Vide, C. (eds.) LATA 2010. LNCS, vol. 6031, pp. 50–57. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13089-2_4
47. Neykova, R., Bocchi, L., Yoshida, N.: Timed runtime monitoring for multiparty conversations. In: FAOC, pp. 1–34 (2017)
48. Neykova, R., Hu, R., Yoshida, N., Abdeljallal, F.: A session type provider: compile-time API generation for distributed protocols with interaction refinements in F#. In: CC 2018. ACM (2018)
49. Neykova, R., Yoshida, N.: Let it recover: multiparty protocol-induced recovery. In: CC 2017, pp. 98–108. ACM (2017)
50. Neykova, R., Yoshida, N.: Multiparty session actors. In: LMCS, pp. 13:1–30 (2017)
51. Ng, N., Yoshida, N.: Static deadlock detection for concurrent Go by global session graph synthesis. In: CC 2016, pp. 174–184 (2016)
52. Ng, N., Yoshida, N., Honda, K.: Multiparty session C: safe parallel programming with message optimisation. In: Furia, C.A., Nanz, S. (eds.) TOOLS 2012. LNCS, vol. 7304, pp. 202–218. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30561-0_15
53. Ocean Observatories Initiative. www.oceanobservatories.org
54. OMG: Business Process Model and Notation (2018). <https://www.omg.org/spec/BPMN/2.0/>
55. Orchard, D.A., Yoshida, N.: Effects as sessions, sessions as effects. In: POPL 2016, pp. 568–581 (2016)
56. Padovani, L.: A simple library implementation of binary sessions. J. Funct. Program. **27**, e4 (2017)
57. Peng, W., Purushothaman, S.: Analysis of a class of communicating finite state machines. Acta Inf. **29**(6/7), 499–522 (1992)
58. Perera, R., Lange, J., Gay, S.J.: Multiparty compatibility for concurrent objects. In: PLACES 2016, pp. 73–82 (2016)

59. Introduction to protocol engineering (2006). <http://cs.uccs.edu/~cs522/pe/pe.htm>
60. Salaün, G., Bordeaux, L., Schaefer, M.: Describing and reasoning on web services using process algebra. IJBPIIM **1**(2), 116–128 (2006)
61. Scalas, A., Dardha, O., Hu, R., Yoshida, N.: A linear decomposition of multiparty sessions for safe distributed programming. In: ECOOP 2017, pp. 24:1–24:31 (2017)
62. Scalas, A., Yoshida, N.: Lightweight session programming in scala. In: ECOOP 2016, pp. 21:1–21:28 (2016)
63. Scalas, A., Yoshida, N.: Less is more: multiparty session types revisited. PACMPL **3**(POPL), 30:1–30:29 (2019)
64. Scribble Project homepage (2018). www.scribble.org
65. Sivaramakrishnan, K.C., Qudeisat, M., Ziarek, L., Nagaraj, K., Eugster, P.: Efficient sessions. Sci. Comput. Program. **78**(2), 147–167 (2013)
66. Takeuchi, K., Honda, K., Kubo, M.: An interaction-based language and its typing system. In: Halatsis, C., Maritsas, D., Philokyprou, G., Theodoridis, S. (eds.) PARLE 1994. LNCS, vol. 817, pp. 398–413. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58184-7_118
67. Taylor, R., Tuosto, E., Walkinshaw, N., Derrick, J.: Choreography-based analysis of distributed message passing programs. In: PDP 2016, pp. 512–519 (2016)
68. Yellin, D.M., Strom, R.E.: Protocol specifications and component adaptors. ACM Trans. Program. Lang. Syst. **19**(2), 292–333 (1997)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.



Security and Hyperproperties



Verifying Hyperliveness

Norine Coenen¹(✉), Bernd Finkbeiner¹,
César Sánchez², and Leander Tentrup¹

¹ Reactive Systems Group, Saarland University,
Saarbrücken, Germany

coenen@react.uni-saarland.de

² IMDEA Software Institute, Madrid, Spain



Abstract. HyperLTL is an extension of linear-time temporal logic for the specification of hyperproperties, i.e., temporal properties that relate multiple computation traces. HyperLTL can express information flow policies as well as properties like symmetry in mutual exclusion algorithms or Hamming distances in error-resistant transmission protocols. Previous work on HyperLTL model checking has focussed on the alternation-free fragment of HyperLTL, where verification reduces to checking a standard trace property over an appropriate self-composition of the system. The alternation-free fragment does, however, not cover general hyperliveness properties. Universal formulas, for example, cannot express the secrecy requirement that for every possible value of a secret variable there exists a computation where the value is different while the observations made by the external observer are the same. In this paper, we study the more difficult case of hyperliveness properties expressed as HyperLTL formulas with quantifier alternation. We reduce existential quantification to strategic choice and show that synthesis algorithms can be used to eliminate the existential quantifiers automatically. We furthermore show that this approach can be extended to reactive system synthesis, i.e., to automatically construct a reactive system that is guaranteed to satisfy a given HyperLTL formula.

1 Introduction

HyperLTL [6] is a temporal logic for *hyperproperties* [7], i.e., for properties that relate multiple computation traces. Hyperproperties cannot be expressed in standard linear-time temporal logic (LTL), because LTL can only express *trace properties*, i.e., properties that characterize the correctness of individual computations. Even branching-time temporal logics like CTL and CTL*, which quantify

This work was partially supported by the German Research Foundation (DFG) as part of the Collaborative Research Center “Foundations of Perspicuous Software Systems” (TRR 248, 389792660), and by the European Research Council (ERC) Grant OSARES (No. 683300), by Madrid Reg. Government project “S2018/TCS-4339 (BLOQUES-CM)”, by EU H2020 project 731535 “Elastest” and by Spanish National Project “BOSCO (PGC2018-102210-B-100)”.

© The Author(s) 2019

I. Dillig and S. Tasiran (Eds.): CAV 2019, LNCS 11561, pp. 121–139, 2019.

https://doi.org/10.1007/978-3-030-25540-4_7

over computation paths, cannot express hyperproperties, because quantifying over a second path automatically means that the subformula can no longer refer to the previously quantified path. HyperLTL addresses this limitation with quantifiers over trace variables, which allow the subformula to refer to all previously chosen traces. For example, *noninterference* [21] between a secret input h and a public output o can be specified in HyperLTL by requiring that all pairs of traces π and π' that always have the same inputs except for h (i.e., all inputs in $I \setminus \{h\}$ are equal on π and π') also have the same output o at all times:

$$\forall \pi. \forall \pi'. \square \left(\bigwedge_{i \in I \setminus \{h\}} i_\pi = i_{\pi'} \right) \Rightarrow \square(o_\pi = o_{\pi'})$$

This formula states that a change in the secret input h alone cannot cause any difference in the output o .

For certain properties of interest, the additional expressiveness of HyperLTL comes at no extra cost when considering the model checking problem. To check a property like noninterference, which only has universal trace quantifiers, one simply builds the self-composition of the system, which provides a separate copy of the state variables for each trace. Instead of quantifying over all pairs of traces, it then suffices to quantify over individual traces of the self-composed system, which can be done with standard LTL. Model checking universal formulas is NLOGSPACE-complete in the size of the system and PSPACE-complete in the size of the formula, which is precisely the same complexity as for LTL.

Universal HyperLTL formulas suffice to express hypersafety properties like noninterference, but not hyperliveness properties that require, in general, quantifier alternation. A prominent example is *generalized noninterference* (GNI) [27], which can be expressed as the following HyperLTL formula:

$$\forall \pi. \forall \pi'. \exists \pi''. \square(h_\pi = h_{\pi''}) \wedge \square(o_{\pi'} = o_{\pi''})$$

This formula requires that for every pair of traces π and π' , there is a third trace π'' in the system that agrees with π on h and with π' on o . The existence of an appropriate trace π'' ensures that in π and π' , the value of o is not determined by the value of h . Generalized noninterference stipulates that low-security outputs may not be altered by the injection of high-security inputs, while permitting nondeterminism in the low-observable behavior. The existential quantifier is needed to allow this nondeterminism. GNI is a hyperliveness property [7] even though the underlying LTL formula is a safety property. The reason for that is that we can extend any set of traces that violates GNI into a set of traces that satisfies GNI, by adding, for each offending pair of traces π, π' , an appropriate trace π'' .

Hyperliveness properties also play an important role in applications beyond security. For example, *robust cleanliness* [9] specifies that significant differences in the output behavior are only permitted after significant differences in the input:

$$\forall \pi. \forall \pi'. \exists \pi''. \square(i_{\pi'} = i_{\pi''}) \wedge (\hat{d}(o_\pi, o_{\pi''}) \leq \kappa_o \text{ } \mathcal{W} \text{ } \hat{d}(i_\pi, i_{\pi''}) > \kappa_i)$$

The differences are measured by a distance function \hat{d} and compared to constant thresholds κ_i for the input and κ_o for the output. The formula specifies

the existence of a trace π'' that globally agrees with π' on the input and where the difference in the output o between π and π'' is bounded by κ_o , unless the difference in the input i between π and π'' was greater than κ_i . Robust cleanliness, thus, forbids unexpected jumps in the system behavior that are, for example, due to software doping, while allowing for behavioral differences due to nondeterminism.

With quantifier alternation, the model checking problem becomes much more difficult. Model checking HyperLTL formulas of the form $\forall^*\exists^*\varphi$, where φ is a quantifier-free formula, is PSPACE-complete in the size of the system and EXPSPACE-complete in the formula. The only known model checking algorithm replaces the existential quantifier with the negation of a universal quantifier over the negated subformula; but this requires a complementation of the system behavior, which is completely impractical for realistic systems.

In this paper, we present an alternative approach to the verification of hyperliveness properties. We view the model checking problem of a formula of the form $\forall\pi.\exists\pi'. \varphi$ as a game between the \forall -player and the \exists -player. While the \forall -player moves through the state space of the system building trace π , the \exists -player must match each move in a separate traversal of the state space resulting in a trace π' such that the pair π, π' satisfies φ . Clearly, the existence of a winning strategy for the \exists -player implies that $\forall\pi.\exists\pi'. \varphi$ is satisfied. The converse is not necessarily true: Even if there always is a trace π' that matches the universally chosen trace π , the \exists -player may not be able to construct this trace, because she only knows about the choices made by the \forall -player in the finite prefix of π that has occurred so far, and not the choices that will be made by the \forall -player in the infinite future. We address this problem by introducing *prophecy variables* into the system. Without changing the behavior of the system, the prophecy variables give the \exists -player the information about the future that is needed to make the right choice after seeing only the finite prefix. Such prophecy variables can be provided manually by the user of the model checker to provide a lookahead on future moves of the \forall -player.

This game-theoretic approach provides an opportunity for the user to reduce the complexity of the model checking problem: If the user provides a strategy for the \exists -player, then the problem reduces to the cheaper model checking problem for universal properties. We show that such strategies can also be constructed automatically using synthesis. Beyond model checking, the game-theoretic approach also provides a method for the synthesis of systems that satisfy a conjunction of hypersafety and hyperliveness properties. Here, we do not only synthesize the strategy, but also construct the system itself, i.e., the game graph on which the model checking game is played. While the synthesis from $\forall^*\exists^*$ hyperproperties is known to be undecidable in general, we show that the game-theoretic approach can naturally be integrated into bounded synthesis, which checks for the existence of a correct system up to a bound on the number of states.

Related Work. While the verification of general HyperLTL formulas has been studied before [6, 17, 18], there has been, so far, no practical model checking algorithm for HyperLTL formulas with quantifier alternation. The existing algorithm involves a complementation of the system automaton, which results in an

exponential blow-up of the state space [18]. The only existing model checker for HyperLTL, MCHYPER [18], was therefore, so far, limited to the alternation-free fragment. Although some hyperliveness properties lie in this fragment, quantifier alternation is needed to express general hyperliveness properties like GNI. In this paper, we present a technique to model check these hyperliveness properties and extend MCHYPER to formulas with quantifier alternation.

The situation is similar in the area of reactive synthesis. There is a synthesis algorithm that automatically constructs implementations from HyperLTL specifications [13] using the bounded synthesis approach [20]. This algorithm is, however, also only applicable to the alternation-free fragment of HyperLTL. In this paper, we extend the bounded synthesis approach to HyperLTL formulas with quantifier alternation. Beyond the model checking and synthesis problems, the satisfiability [11, 12, 14] and monitoring [15, 16, 22] problems of HyperLTL have also been studied in the past.

For certain information-flow security policies, there are verification techniques that use methods related to our model checking and synthesis algorithms. Specifically, the self-composition technique [2, 3], a construction based on the product of copies of a system, has been tailored for various trace-based security definitions [10, 23, 28]. Unlike our algorithms, these techniques focus on specific information-flow policies, not on a general logic like HyperLTL.

The use of prophecy variables [1] to make information about the future accessible is a known technique in the verification of trace properties. It is, for example, used to establish simulation relations between automata [26] or in the verification of CTL* properties [8].

In our game-theoretic view on the model checking problem for $\forall^*\exists^*$ hyperproperties the \exists -player has an infinite lookahead. There is some work on *finite* lookahead on trace languages [24]. We use the idea of finite lookahead as an approximation to construct existential strategies and give a novel synthesis construction for strategies with delay based on bounded synthesis [20].

2 Preliminaries

For tuples $\mathbf{x} \in X^n$ and $\mathbf{y} \in X^m$ over set X , we use $\mathbf{x} \cdot \mathbf{y} \in X^{n+m}$ to denote the concatenation of \mathbf{x} and \mathbf{y} . Given a function $f: X \rightarrow Y$ and a tuple $\mathbf{x} \in X^n$, we define by $f \circ \mathbf{x} \in Y^n$ the tuple $(f(\mathbf{x}[1]), \dots, f(\mathbf{x}[n]))$. Let AP be a finite set of atomic propositions and let $\Sigma = 2^{\text{AP}}$ be the corresponding alphabet. A *trace* $t \in \Sigma^\omega$ is an infinite sequence of elements of Σ . We denote a set of traces by $Tr \subseteq \Sigma^\omega$. We define $t[i, \infty]$ to be the suffix of t starting at position $i \geq 0$.

HyperLTL. HyperLTL [6] is a temporal logic for specifying hyperproperties. It extends LTL by quantification over trace variables π and a method to link atomic propositions to specific traces. Let \mathcal{V} be an infinite set of trace variables. Formulas in HyperLTL are given by the grammar

$$\begin{aligned}\varphi &:= \forall\pi.\varphi \mid \exists\pi.\varphi \mid \psi, \text{ and} \\ \psi &:= a_\pi \mid \neg\psi \mid \psi \vee \psi \mid \bigcirc\psi \mid \psi \mathcal{U} \psi,\end{aligned}$$

where $a \in AP$ and $\pi \in \mathcal{V}$. We allow the standard boolean connectives $\wedge, \rightarrow, \leftrightarrow$ as well as the derived LTL operators release $\varphi \mathcal{R} \psi \equiv \neg(\neg\varphi \mathcal{U} \neg\psi)$, eventually $\Diamond\varphi \equiv \text{true } \mathcal{U} \varphi$, globally $\Box\varphi \equiv \neg\Diamond\neg\varphi$, and weak until $\varphi \mathcal{W} \psi \equiv \Box\varphi \vee (\varphi \mathcal{U} \psi)$.

We call a $\mathcal{Q}^+ \mathcal{Q}'^+ \varphi$ HyperLTL formula (for $\mathcal{Q}, \mathcal{Q}' \in \{\forall, \exists\}$) and quantifier-free formula φ) *alternation-free* iff $\mathcal{Q} = \mathcal{Q}'$. Further, we say that $\mathcal{Q}^+ \mathcal{Q}'^+ \varphi$ has *one quantifier alternation* (or lies in the *one-alternation fragment*) iff $\mathcal{Q} \neq \mathcal{Q}'$.

The semantics of HyperLTL is given by the satisfaction relation \models_{Tr} over a set of traces $Tr \subseteq \Sigma^\omega$. We define an assignment $\Pi : \mathcal{V} \rightarrow \Sigma^\omega$ that maps trace variables to traces. $\Pi[\pi \mapsto t]$ updates Π by assigning variable π to trace t .

$\Pi, i \models_{Tr} a_\pi$	iff $a \in \Pi(\pi)[i]$
$\Pi, i \models_{Tr} \neg\varphi$	iff $\Pi, i \not\models_{Tr} \varphi$
$\Pi, i \models_{Tr} \varphi \vee \psi$	iff $\Pi, i \models_{Tr} \varphi$ or $\Pi, i \models_{Tr} \psi$
$\Pi, i \models_{Tr} \bigcirc \varphi$	iff $\Pi, i + 1 \models_{Tr} \varphi$
$\Pi, i \models_{Tr} \varphi \mathcal{U} \psi$	iff $\exists j \geq i. \Pi, j \models_{Tr} \psi \wedge \forall i \leq k < j. \Pi, k \models_{Tr} \varphi$
$\Pi, i \models_{Tr} \exists \pi. \varphi$	iff there is some $t \in Tr$ such that $\Pi[\pi \mapsto t], i \models_{Tr} \varphi$
$\Pi, i \models_{Tr} \forall \pi. \varphi$	iff for all $t \in Tr$ it holds that $\Pi[\pi \mapsto t], i \models_{Tr} \varphi$

We write $Tr \models \varphi$ for $\{\} \models_{Tr} \varphi$ where $\{\}$ denotes the empty assignment.

Every hyperproperty is an intersection of a hypersafety and a hyperliveness property [7]. A *hypersafety* property is one where there is a finite set of finite traces that is a bad prefix, i.e., that cannot be extended into a set of traces that satisfies the hypersafety property. A *hyperliveness* property is a property where every finite set of finite traces can be extended to a possibly infinite set of infinite traces such that the resulting trace set satisfies the hyperliveness property.

Transition Systems. We use transition systems as a model of computation for reactive systems. Transition systems consume sequences over an input alphabet by transforming their internal state in every step. Let I and O be a finite set of input and output propositions, respectively, and let $\Upsilon = 2^I$ and $\Gamma = 2^O$ be the corresponding finite alphabets. A Γ -labeled Υ -transition system \mathcal{S} is a tuple $\langle S, s_0, \tau, l \rangle$, where S is a finite set of states, $s_0 \in S$ is the designated initial state, $\tau : S \times \Upsilon \rightarrow S$ is the transition function, and $l : S \rightarrow \Gamma$ is the state-labeling function. We write $s \xrightarrow{v} s'$ or $(s, v, s') \in \tau$ if $\tau(s, v) = s'$. We generalize the transition function to sequences over Υ by defining $\tau^* : \Upsilon^* \rightarrow S$ recursively as $\tau^*(\epsilon) = s_0$ and $\tau^*(v_0 \cdots v_{n-1} v_n) = \tau(\tau^*(v_0 \cdots v_{n-1}), v_n)$ for $v_0 \cdots v_{n-1} v_n \in \Upsilon^+$. Given an infinite word $v = v_0 v_1 \dots \in \Upsilon^\omega$, the transition system produces an infinite sequence of outputs $\gamma = \gamma_0 \gamma_1 \gamma_2 \dots \in \Gamma^\omega$, such that $\gamma_i = l(\tau^*(v_0 \dots v_{i-1}))$ for every $i \geq 0$. The resulting trace ρ is $(v_0 \cup \gamma_0)(v_1 \cup \gamma_1) \dots \in \Sigma^\omega$ where we have $AP = I \cup O$. The set of traces generated by \mathcal{S} is denoted by $traces(\mathcal{S})$. Furthermore, we define $\varepsilon = \langle \{s\}, s, \tau_\varepsilon, l_\varepsilon \rangle$ as the transition system over $I = O = \emptyset$ that has only a single trace, that is $traces(\varepsilon) = \{\emptyset^\omega\}$. For this transition system, $\tau_\varepsilon(s, \emptyset) = s$ and $l_\varepsilon(s) = \emptyset$. Given two transition systems $\mathcal{S} = \langle S, s_0, \tau, l \rangle$ and $\mathcal{S}' = \langle S', s'_0, \tau', l' \rangle$, we define $\mathcal{S} \times \mathcal{S}' = \langle S \times S', (s_0, s'_0), \tau'', l'' \rangle$ as the Γ^2 -labeled Υ^2 -transition system where $\tau''((s, s'), (v, v')) = (\tau(s, v), \tau'(s', v'))$ and $l''((s, s')) = (l(s), l'(s'))$. A transition system \mathcal{S} satisfies a general HyperLTL formula φ , if, and only if, $traces(\mathcal{S}) \models \varphi$.

Automata. An alternating parity automaton \mathcal{A} over a finite alphabet Σ is a tuple $\langle Q, q_0, \delta, \alpha \rangle$, where Q is a finite set of states, $q_0 \in Q$ is the designated initial state, $\delta: Q \times \Sigma \rightarrow \mathbb{B}^+(Q)$ is the transition function, and $\alpha: Q \rightarrow C$ is a function that maps states of \mathcal{A} to a finite set of colors $C \subset \mathbb{N}$. For $C = \{0, 1\}$ and $C = \{1, 2\}$, we call \mathcal{A} a co-Büchi and Büchi automaton, respectively, and we use the sets $F \subseteq Q$ and $B \subseteq Q$ to represent the rejecting ($C = 1$) and accepting ($C = 2$) states in the respective automaton (as a replacement of the coloring function α). A safety automaton is a Büchi automaton where every state is accepting. The transition function δ maps a state $q \in Q$ and some $a \in \Sigma$ to a positive Boolean combination of successor states $\delta(q, a)$. An automaton is *non-deterministic* or *universal* if δ is purely disjunctive or conjunctive, respectively.

A run of an alternating automaton is a Q -labeled tree. A tree T is a subset of $\mathbb{N}_{>0}^*$ such that for every node $n \in \mathbb{N}_{>0}^*$ and every positive integer $i \in \mathbb{N}_{>0}$, if $n \cdot i \in T$ then (i) $n \in T$ (i.e., T is prefix-closed), and (ii) for every $0 < j < i$, $n \cdot j \in T$. The root of T is the empty sequence ϵ and for a node $n \in T$, $|n|$ is the length of the sequence n , in other words, its distance from the root. A run of \mathcal{A} on an infinite word $\rho \in \Sigma^\omega$ is a Q -labeled tree (T, r) such that $r(\epsilon) = q_0$ and for every node $n \in T$ with children n_1, \dots, n_k the following holds: $1 \leq k \leq |Q|$ and $\{r(n_1), \dots, r(n_k)\} \models \delta(q, \rho[i])$, where $q = r(n)$ and $i = |n|$. A path is accepting if the highest color appearing infinitely often is even. A run is accepting if all its paths are accepting. The language of \mathcal{A} , written $\mathcal{L}(\mathcal{A})$, is the set $\{\rho \in \Sigma^\omega \mid \mathcal{A} \text{ accepts } \rho\}$. A transition system \mathcal{S} is accepted by an automaton \mathcal{A} , written $\mathcal{S} \models \mathcal{A}$, if $\text{traces}(\mathcal{S}) \subseteq \mathcal{L}(\mathcal{A})$.

Strategies. Given two disjoint finite alphabets Υ and Γ , a strategy $\sigma: \Upsilon^* \rightarrow \Gamma$ is a mapping from finite histories of Υ to Γ . A transition system $\mathcal{S} = \langle S, s_0, \tau, l \rangle$ generates the strategy σ if $\sigma(\mathbf{v}) = l(\tau^*(\mathbf{v}))$ for every $\mathbf{v} \in \Upsilon^*$. A strategy σ is called *finite-state* if there exists a transition system that generates σ .

In the following, we use finite-state strategies to modify the inputs of transition systems. Let $\mathcal{S} = \langle S, s_0, \tau, l \rangle$ be a transition system over input and output alphabets Υ and Γ and let $\sigma: (\Upsilon')^* \rightarrow \Upsilon$ be a finite-state strategy. Let $\mathcal{S}' = \langle S', s'_0, \tau', l' \rangle$ be the transition system implementing σ , then $\mathcal{S} \parallel \sigma = \mathcal{S} \parallel \mathcal{S}'$ is the transition system $\langle S \times S', (s_0, s'_0), \tau^{\parallel}, l^{\parallel} \rangle$ where $\tau^{\parallel}: (S \times S') \times \Upsilon' \rightarrow (S \times S')$ is defined as $\tau^{\parallel}((s, s'), v') = (\tau(s, l'(s')), \tau'(s', v'))$ and $l^{\parallel}: (S \times S') \rightarrow \Gamma$ is defined as $l^{\parallel}(s, s') = l(s)$ for every $s \in S$, $s' \in S'$, and $v' \in \Upsilon'$.

Model Checking HyperLTL. We recap the model checking of universal HyperLTL formulas. This case, as well as the dual case of only existential quantifiers, is well-understood and, in fact, efficiently implemented in the model checker MCHYPER [18]. The principle behind the model checking approach is *self-composition*, where we check a standard trace property on a composition of an appropriate number of copies of the given system.

Let *zip* denote the function that maps an n -tuple of sequences to a single sequence of n -tuples, for example, $\text{zip}([1, 2, 3], [4, 5, 6]) = [(1, 4), (2, 5), (3, 6)]$, and let *unzip* denote its inverse. Given $\mathcal{S} = \langle S, s_0, \tau, l \rangle$, the n -fold self-composition of \mathcal{S} is the transition system $\mathcal{S}^n = \langle S^n, s'_0, \tau_n, l_n \rangle$, where $s'_0 := (s_0, \dots, s_0) \in S^n$, $\tau_n(\mathbf{s}, \mathbf{v}) := \tau \circ \text{zip}(\mathbf{s}, \mathbf{v})$ and $l_n(\mathbf{s}) := l \circ s$ for every $\mathbf{s} \in S^n$ and $\mathbf{v} \in \Upsilon^n$. If $\text{traces}(\mathcal{S})$

is the set of traces generated by \mathcal{S} , then $\{\text{zip}(\rho_1, \dots, \rho_n) \mid \rho_1, \dots, \rho_n \in \text{traces}(\mathcal{S})\}$ is the set of traces generated by \mathcal{S}^n . We use the notation $\text{zip}(\varphi, \pi_1, \pi_2, \dots, \pi_n)$ for some HyperLTL formula φ to combine the trace variables $\pi_1, \pi_2, \dots, \pi_n$ (occurring free in φ) into a fresh trace variable π^* .

Theorem 1 (Self-composition for universal HyperLTL formulas [18]). *For a transition system \mathcal{S} and a HyperLTL formula of the form $\forall \pi_1. \forall \pi_2. \dots. \forall \pi_n. \varphi$ it holds that $\mathcal{S} \models \forall \pi_1. \forall \pi_2. \dots. \forall \pi_n. \varphi$ iff $\mathcal{S}^n \models \forall \pi^*. \text{zip}(\varphi, \pi_1, \pi_2, \dots, \pi_n)$.*

Theorem 2 (Complexity of model checking universal formulas [18]). *The model checking problem for universal HyperLTL formulas is PSPACE-complete in the size of the formula and NLOGSPACE-complete in the size of the transition system.*

The complexity of verifying universal HyperLTL formulas is exactly the same as the complexity of verifying LTL formulas. For HyperLTL formulas with quantifier alternations, the model checking problem is significantly more difficult.

Theorem 3 (Complexity of model checking formulas with one quantifier alternation [18]). *The model checking problem for HyperLTL formulas with one quantifier alternation is in EXPSPACE in the size of the formula and in PSPACE in the size of the transition system.*

One way to circumvent this complexity is to fix the existential choice and strengthen the formula to the universal fragment [9, 13, 18]. While avoiding the complexity problem, this transformation requires deep knowledge of the system, is prone to errors, and cannot be verified automatically as the problem of checking implications becomes undecidable [11]. In the following section, we present a technique that circumvents the complexity problem while still inheriting strong correctness guarantees. Further, we provide a method that can, under certain restrictions, derive a strategy for the existential choice automatically.

3 Model Checking with Quantifier Alternations

3.1 Model Checking with Given Strategies

Our first goal is the verification of HyperLTL formulas with one quantifier alternation, i.e., formulas of the form $\forall^* \exists^* \varphi$ or $\exists^* \forall^* \varphi$, where φ is a quantifier-free formula. Note that the presented techniques can, similar to skolemization, be extended to more than one quantifier alternation. Quantifier alternation introduces dependencies between the quantified traces. In a $\forall^* \exists^* \varphi$ formula, the choices of the existential quantifiers depend on the choices of the universal quantifiers preceding them. In a formula of the form $\exists^* \forall^* \varphi$, however, there has to be a single choice for the existential quantifiers that works for all choices of the universal quantifiers. In this case, the existentially quantified variables do not depend on the universally quantified variables. Hence, the witnesses for the existential quantifiers are traces rather than functions that map tuples of traces

to traces. As established above, the model checking problem for HyperLTL formulas with quantifier alternation is known to be significantly more difficult than the model checking problem for universal formulas.

Our verification technique for formulas with quantifier alternation is to substitute strategic choice for existential choice. As discussed in the introduction, the existence of a strategy implies the existence of a trace.

Theorem 4 (Substituting Strategic Choice for Existential Choice). *Let \mathcal{S} be a transition system over input alphabet Υ .*

It holds that $\mathcal{S} \models \forall\pi_1\forall\pi_2\dots\forall\pi_n. \exists\pi'_1\exists\pi'_2\dots\exists\pi'_m. \varphi$ if there is a strategy $\sigma : (\Upsilon^n)^ \rightarrow \Upsilon^m$ such that $\mathcal{S}^n \times (\mathcal{S}^m \parallel \sigma) \models \forall\pi^*. \text{zip}(\varphi, \pi_1, \pi_2, \dots, \pi_n, \pi'_1, \pi'_2, \dots, \pi'_m)$.*

It holds that $\mathcal{S} \models \exists\pi_1\exists\pi_2\dots\exists\pi_m. \forall\pi'_1\forall\pi'_2\dots\forall\pi'_n. \varphi$ if there is a strategy $\sigma : (\Upsilon^0)^ \rightarrow \Upsilon^m$ such that $(\mathcal{S}^m \parallel \sigma) \times \mathcal{S}^n \models \forall\pi^*. \text{zip}(\varphi, \pi_1, \pi_2, \dots, \pi_m, \pi'_1, \pi'_2, \dots, \pi'_n)$.*

Proof. Let σ be such a strategy, then we define a witness for the existential trace quantifiers $\exists\pi'_1\exists\pi'_2\dots\exists\pi'_m$ as the sequence of inputs $v = v_0v_1\dots \in (\Upsilon^m)^\omega$ such that $v_i = \sigma(v'_0v'_1\dots v'_{i-1})$ for every $i \geq 0$ and every $v'_i \in \Upsilon^n$; analogously, we define a witness for the existential trace quantifiers $\exists\pi_1\exists\pi_2\dots\exists\pi_m$ as the sequence of inputs $v = v_0v_1\dots \in (\Upsilon^m)^\omega$ such that $v_i = \sigma(v'_0v'_1\dots v'_{i-1})$ for every $i \geq 0$ and every $v'_i \in \Upsilon^0$. \square

An application of the theorem reduces the verification problem of a HyperLTL formula with one quantifier alternation to the verification problem of a universal HyperLTL formula. If a sufficiently small strategy can be found, the reduction in complexity is substantial:

Corollary 1 (Model checking with Given Strategies). *The model checking problem for HyperLTL formulas with one quantifier alternation and given strategies for the existential quantifiers is in PSPACE in the size of the formula and NLOGSPACE in the size of the product of the strategy and the system.*

Note that the converse of Theorem 4 is not in general true. The satisfaction of a $\forall^*\exists^*$ HyperLTL formula does not imply the existence of a strategy, because at any given point in time the strategy only knows about a finite prefix of the universally quantified traces. Consider the formula $\forall\pi\exists\pi'. \bigcirc a_\pi \leftrightarrow a_{\pi'}$ and a system that can produce arbitrary sequences of a and $\neg a$. Although the system satisfies the formula, it is not possible to give a strategy that allows us to prove this fact. Whatever choice our strategy makes, the next move of the \forall -player can make sure that the strategy's choice was wrong. In the following, we present a method that addresses this problem.

Prophecy Variables. A classic technique for resolving future dependencies is the introduction of *prophecy variables* [1]. Prophecy variables are auxiliary variables that are added to the system without affecting the behavior of the system. Such variables can be used to make predictions about the future.

We use prophecy variables to define strategies that depend on the future. In the example discussed above, $\forall\pi\exists\pi'. \bigcirc a_\pi \leftrightarrow a_{\pi'}$, the choice of the value of $a_{\pi'}$ in

the first position depends on the value of a_π in the second position. We introduce a prophecy variable p that predicts in the first position whether a_π is true in the second position. With the prophecy variable, there exists a strategy that correctly assigns the value of p whenever the prediction is correct: The strategy chooses to set $a_{\pi'}$ if, and only if, p holds.

Technically, the proof technique introduces a set of fresh input variables P into the system. For a Γ -labeled Υ -transition system $\mathcal{S} = \langle S, s_0, \tau, l \rangle$, we define the Γ -labeled $(\Upsilon \cup P)$ -transition system $\mathcal{S}^P = \langle S, s_0, \tau^P, l \rangle$ including the inputs P where $\tau^P : S \times (\Upsilon \cup P) \rightarrow S$. For all $s \in S$ and $v^P \in \Upsilon \cup P$, $\tau^P(s, v^P) = \tau(s, v)$ for $v \in \Upsilon$ obtained by removing the variables in P from v^P (i.e., $v =_{\setminus P} v^P$). Moreover, the proof technique modifies the specification so that the original property only needs to be satisfied if the prediction is actually correct. We obtain the modified specification $\forall \pi \exists \pi'. (p_\pi \leftrightarrow \bigcirc a_\pi) \rightarrow (\bigcirc a_\pi \leftrightarrow a_{\pi'})$ in our example. The following theorem describes the general technique for one prophecy variable.

Theorem 5 (Model checking with Prophecy Variables). *For a transition system \mathcal{S} and a quantifier-free formula φ , let ψ be a quantifier-free formula over the universally quantified trace variables $\pi_1, \pi_2 \dots \pi_n$ and let p be a fresh atomic proposition. It holds that $\mathcal{S} \models \forall \pi_1 \forall \pi_2 \dots \forall \pi_n. \exists \pi'_1 \exists \pi'_2 \dots \exists \pi'_m. \varphi$ if, and only if, $\mathcal{S}^{\{p\}} \models \forall \pi_1 \forall \pi_2 \dots \forall \pi_n. \exists \pi'_1 \exists \pi'_2 \dots \exists \pi'_m. \square(p_{\pi_1} \leftrightarrow \psi) \rightarrow \varphi$.*

Note that ψ is restricted to refer only to *universally* quantified trace variables. Without this restriction, the method would not be sound. In our example, $\psi = a_{\pi'}$ would lead to the modified formula $\forall \pi \exists \pi'. (p_\pi \leftrightarrow a_{\pi'}) \rightarrow (\bigcirc a_\pi \leftrightarrow a_{\pi'})$, which could be satisfied with the strategy that assigns $a_{\pi'}$ to *true* iff p_π is *false*, and thus falsifies the assumption that the prediction is correct, rather than ensuring that the original formula is true.

Proof. It is easy to see that the original specification implies the modified specification, since the original formula is the conclusion of the implication. Assume that the modified specification holds. Since the prophecy variable p is a fresh atomic proposition, and ψ does not refer to the existentially chosen traces, we can, for every choice of the universally quantified traces, always choose the value of p such that it guesses correctly, i.e., that p is *true* whenever ψ holds. In this case, the conclusion and therefore the original specification must be *true*. \square

Unfortunately, prophecy variables do not provide a complete proof technique. Consider a system allowing arbitrary sequences of a and b and this specification:

$$\begin{aligned} & \forall \pi \exists \pi'. b_{\pi'} \wedge \square(b_{\pi'} \leftrightarrow \bigcirc \neg b_{\pi'}) \\ & \quad \wedge (a_{\pi'} \rightarrow (a_\pi \mathcal{W} (b_{\pi'} \wedge \neg a_\pi))) \\ & \quad \wedge (\neg a_{\pi'} \rightarrow (a_\pi \mathcal{W} (\neg b_{\pi'} \wedge \neg a_\pi))) \end{aligned}$$

Intuitively, π' has to be able to predict whether π will stop outputting a at an even or odd position of the trace. There is no HyperLTL formula to be used as ψ in Theorem 5, because, like LTL, HyperLTL can only express non-counting properties. It is worth noting that in our practical experiments, the

incompleteness was never a problem. In many cases, it is not even necessary to add prophecy variables at all. The presented proof technique is, thus, practically useful despite this incompleteness result.

3.2 Model Checking with Synthesized Strategies

We now extend the model checking approach with the automatic synthesis of the strategies for the existential quantifiers. For a given HyperLTL formula of the form $\forall^n \exists^m \varphi$ and a transition system \mathcal{S} , we search for a transition system $\mathcal{S}_\exists = \langle X, x_0, \mu, l_\exists \rangle$, where X is a set of states, $x_0 \in X$ is the designated initial state, $\mu: X \times \Upsilon^n \rightarrow X$ is the transition function, and $l_\exists: X \rightarrow \Upsilon^m$ is the labeling function, such that $\mathcal{S}^n \times (\mathcal{S}^m \parallel \mathcal{S}_\exists) \models \text{zip}(\varphi)$. (Since for formulas of the form $\exists^m \forall^n \varphi$ the problem only differs in the input of \mathcal{S}_\exists , we focus on $\forall \exists$ HyperLTL.)

Theorem 6. *The strategy realizability problem for $\forall^* \exists^*$ formulas is 2EXPTIME-complete.*

Proof (Sketch). We reduce the strategy synthesis problem to the problem of synthesizing a distributed reactive system with a single black-box process. This problem is decidable [19] and can be solved in 2EXPTIME. The lower bound follows from the LTL realizability problem [30]. \square

The decidability result implies that there is an upper bound on the size of \mathcal{S}_\exists that is doubly exponential in φ . Thus, the bounded synthesis approach [20] can be used to search for increasingly larger implementations, until a solution is found or the maximal bound is reached, yielding an efficient decision procedure for the strategy synthesis problem. In the following, we describe this approach in detail.

Bounded Synthesis of Strategies. We transform the synthesis problem into an SMT constraint satisfaction problem, where we leave the representation of strategies uninterpreted and challenge the solver to provide an interpretation. Given a HyperLTL formula $\forall^n \exists^m \varphi$ where φ is quantifier-free, the model checking is based on the product of the n -fold self composition of the transition system \mathcal{S} , the m -fold self-composition of \mathcal{S} where the strategy \mathcal{S}_\exists controls the inputs, and the universal co-Büchi automaton \mathcal{A}_φ representing the language $\mathcal{L}(\varphi)$ of φ .

For a quantifier-free HyperLTL formula φ , we construct the universal co-Büchi automaton \mathcal{A}_φ such that $\mathcal{L}(\mathcal{A}_\varphi)$ is the set of words w such that $\text{unzip}(w) \models \varphi$, i.e., the tuple of traces satisfies φ . We get this automaton by dualizing the non-deterministic Büchi automaton for $\neg\psi$ [6], i.e., changing the branching from non-deterministic to universal and the acceptance condition from Büchi to co-Büchi. Hence, \mathcal{S} satisfies a universal HyperLTL formula $\forall \pi_1 \dots \forall \pi_n. \varphi$ if the traces generated by the self-composition \mathcal{S}^n are a subset of $\mathcal{L}(\mathcal{A}_\varphi)$.

In more detail, the algorithm searches for a transition system $\mathcal{S}_\exists = \langle X, x_0, \mu, l_\exists \rangle$ such that the run graph of \mathcal{S}^n , $\mathcal{S}^m \parallel \mathcal{S}_\exists$, and \mathcal{A}_φ , written $\mathcal{S}^n \times (\mathcal{S}^m \parallel \mathcal{S}_\exists) \times \mathcal{A}_\varphi$, is accepting. Formally, given a Γ -labeled Υ -transition

system $\mathcal{S} = \langle S, s_0, \tau, l \rangle$ and a universal co-Büchi automaton $\mathcal{A}_\varphi = \langle Q, q_0, \delta, F \rangle$, where $\delta: Q \times \mathcal{Y}^{n+m} \times \Gamma^{n+m} \rightarrow 2^Q$, the run graph $\mathcal{S}^n \times (\mathcal{S}^m \parallel \mathcal{S}_\exists) \times \mathcal{A}_\varphi$ is the directed graph (V, E) , with the set of vertices $V = \mathcal{S}^n \times \mathcal{S}^m \times X \times Q$, initial vertex $v_{init} = ((s_0, \dots, s_0), (s_0, \dots, s_0), x_0, q_0)$ and the edge relation $E \subseteq V \times V$ satisfying $((\mathbf{s}_n, \mathbf{s}_m, x, q), (\mathbf{s}'_n, \mathbf{s}'_m, x', q')) \in E$ if, and only if

$$\begin{aligned} \exists \mathbf{v} \in \mathcal{Y}^n. & \left(\mathbf{s}_n \xrightarrow[\tau_n]{\mathbf{v}} \mathbf{s}'_n \right) \wedge \left(\mathbf{s}_m \xrightarrow[\tau_m]{l_\exists(x)} \mathbf{s}'_m \right) \wedge \left(x \xrightarrow[\mu]{\mathbf{v}} x' \right) \\ & \wedge q' \in \delta(q, \mathbf{v} \cdot l_\exists(x), l_n(\mathbf{s}_n) \cdot l_m(\mathbf{s}_m)). \end{aligned}$$

Theorem 7. *Given \mathcal{S} , \mathcal{S}_\exists , and a HyperLTL formula $\forall^n \exists^m \varphi$ where φ is quantifier-free. Let \mathcal{A}_φ be the universal co-Büchi automaton for φ . If the run graph $\mathcal{S}^n \times (\mathcal{S}^m \parallel \mathcal{S}_\exists) \times \mathcal{A}_\varphi$ is accepting, then $\mathcal{S} \models \forall^n \exists^m \varphi$.*

Proof. Follows from Theorem 4 and the fact that \mathcal{A}_φ represents $\mathcal{L}(\varphi)$. \square

The acceptance of a run graph is witnessed by an annotation $\lambda: V \rightarrow \mathbb{N} \cup \{\perp\}$ which is a function mapping every reachable vertex $v \in V$ in the run graph to a natural number $\lambda(v)$, i.e., $\lambda(v) \neq \perp$. Intuitively, $\lambda(v)$ returns the number of visits to rejecting states on any path from the initial vertex v_{init} to v . If we can bound this number for every reachable vertex, the annotation is *valid* and the run graph is accepting. Formally, an annotation λ is valid, if (1) the initial state is reachable ($\lambda(v_{init}) \neq \perp$) and (2) for every $(v, v') \in E$ with $\lambda(v) \neq \perp$ it holds that $\lambda(v') \neq \perp$ and $\lambda(v) > \lambda(v')$ where \geq is $>$ if v' is rejecting and \geq otherwise. Such an annotation exists if, and only if, the run graph is accepting [20].

We encode the search for \mathcal{S}_\exists and the annotation λ as an SMT constraint system. Therefore, we use uninterpreted function symbols to encode \mathcal{S}_\exists and λ . A transition system \mathcal{S} is represented in the constraint system by two functions, the transition function $\tau: S \times \mathcal{Y} \rightarrow S$ and the labeling function $l: S \rightarrow \Gamma$. The annotation is split into two parts, a reachability constraint $\lambda^\mathbb{B}: V \rightarrow \mathbb{B}$ indicating whether a state in the run graph is reachable and a counter $\lambda^\# : V \rightarrow \mathbb{N}$ that maps every reachable vertex v to the maximal number of rejecting states $\lambda^\#(v)$ visited by any path from the initial vertex to v . The resulting constraint asserts that there is a transition system \mathcal{S}_\exists with an accepting run graph. Note, that the functions representing the system \mathcal{S} ($\tau: S \times \mathcal{Y} \rightarrow S$ and $l: S \rightarrow \Gamma$) are given, that is, they are interpreted.

$$\begin{aligned} \exists \lambda^\mathbb{B}: & S^n \times S^m \times X \times Q \rightarrow \mathbb{B}. \exists \lambda^\# : S^n \times S^m \times X \times Q \rightarrow \mathbb{N}. \\ \exists \mu: & X \times \mathcal{Y}^n \rightarrow X. \exists l_\exists: X \rightarrow \mathcal{Y}^m \\ \forall \mathbf{v} \in \mathcal{Y}^n. \forall & \mathbf{s}_n, \mathbf{s}'_n \in S^n. \forall \mathbf{s}_m, \mathbf{s}'_m \in S^m. \forall q, q' \in Q. \forall x, x' \in X. \\ \lambda^\mathbb{B}((s_0, \dots, s_0), & (s_0, \dots, s_0), x_0, q_0) \wedge \\ \left(\lambda^\mathbb{B}(\mathbf{s}_n, \mathbf{s}_m, x, q) \wedge q' \in \delta(q, (\mathbf{v} \cdot l_\exists(x)), (l \circ (\mathbf{s}_n \cdot \mathbf{s}_m))) \wedge x' = \mu(x, \mathbf{v}) \right. \\ \wedge \mathbf{s}'_n = \tau_n(\mathbf{s}_n, \mathbf{v}) \wedge \mathbf{s}'_m = \tau_m(\mathbf{s}_m, l_\exists(x))) \Big) \\ \Rightarrow \lambda^\mathbb{B}(\mathbf{s}'_n, \mathbf{s}'_m, x', q') \wedge \lambda^\#(\mathbf{s}_n, \mathbf{s}_m, x, q) \geq \lambda^\#(\mathbf{s}'_n, \mathbf{s}'_m, x', q') \end{aligned}$$

where \triangleright is $>$ if $q' \in F$ and \geq otherwise. The *bounded synthesis algorithm* increases the bound of the strategy \mathcal{S}_\exists until either the constraints system becomes satisfiable, or a given upper bound is reached. In the case the constraint system is satisfiable, we can extract interpretations for the functions μ and l_\exists using a solver that is able to produce models. These functions then represent the synthesized transition system \mathcal{S}_\exists .

Corollary 2. *Given \mathcal{S} and a HyperLTL formula $\forall^*\exists^*\varphi$ where φ is quantifier-free. If the constraint system is satisfiable for some bound on the size of \mathcal{S}_\exists then $\mathcal{S} \models \forall^*\exists^*\varphi$.*

Proof. Follows immediately by Theorem 7. \square

As the decision problem is decidable, we know that there is an upper bound on the size of a realizing \mathcal{S}_\exists and, thus, the bounded synthesis approach is a decision procedure for the strategy realizability problem.

Corollary 3. *The bounded synthesis algorithm decides the strategy realizability problem for $\forall^*\exists^*$ HyperLTL.*

Proof. The existence of such an upper bound follows from Theorem 6. \square

Approximating Prophecy. We introduce a new parameter to the strategy synthesis problem to approximate the information about the future that can be captured using prophecy variables. This bound represents a constant *lookahead* into future choices made by the environment. In other words, for a given $k \geq 0$, the strategy \mathcal{S}_\exists is allowed to depend on choices of the \forall -player in the next k steps. While constant lookahead is only an approximation of infinite clairvoyance, it suffices for many practical situations as shown by prior case studies [9, 18].

We present a solution to synthesizing transition systems with constant lookahead for $k \geq 0$ using bounded synthesis. To simplify the presentation, we present the stand-alone problem with respect to a specification given as a universal co-Büchi automaton. The integration into the constraint system for the $\forall^*\exists^*$ HyperLTL synthesis as presented in the previous section is then straightforward. First, we present an extension to the transition system model that incorporates the notion of constant lookahead. The idea of this extension is to replace the initial state s_0 by a function $init: \Upsilon^k \rightarrow S$ that maps input sequences of length k to some state. Thus, the transition system observes the first k inputs, chooses some initial state based on those inputs, and then progresses with the same pace as the input sequence. Next, we define the run graph of such a system $\mathcal{S}_k = \langle S, init, \tau, l \rangle$ and an automaton $\mathcal{A} = \langle Q, q_0, \delta, F \rangle$, where $\delta: Q \times \Upsilon \times \Gamma \rightarrow Q$, as the directed graph (V, E) with the set of vertices $V = S \times Q \times \Upsilon^k$, the initial vertices $(s, q_0, \mathbf{v}) \in V$ such that $s = init(\mathbf{v})$ for every $\mathbf{v} \in \Upsilon^k$, and the edge relation $E \subseteq V \times V$ satisfying $((s, q, v_1 v_2 \cdots v_k), (s', q', v'_1 v'_2 \cdots v'_k)) \in E$ if, and only if

$$\exists v_{k+1} \in \Upsilon. s \xrightarrow{v_{k+1}} s' \wedge q' \in \delta(q, v_1, l(s)) \wedge \bigwedge_{1 \leq i \leq k} v'_i = v_{i+1}.$$

Lemma 1. *Given a universal co-Büchi automaton \mathcal{A} and a k -lookahead transition system \mathcal{S}_k . $\mathcal{S}_k \models \mathcal{A}$ if, and only if, the run graph $\mathcal{S}_k \times \mathcal{A}$ is accepting.*

Finally, synthesis amounts to solving the following constraint system:

$$\begin{aligned} & \exists \lambda^{\mathbb{B}}: S \times Q \times \Upsilon^k \rightarrow \mathbb{B}. \exists \lambda^{\mathbb{N}}: S \times Q \times \Upsilon^k \rightarrow \mathbb{N}. \\ & \exists \text{init}: \Upsilon^k \rightarrow S. \exists \tau: S \times \Upsilon \rightarrow S. \exists l: S \rightarrow \Gamma. \\ & (\forall \mathbf{v} \in \Upsilon^k. \lambda^{\mathbb{B}}(\text{init}(\mathbf{v}), q_0, \mathbf{v})) \wedge \\ & \forall v_1 v_2 \dots v_{k+1} \in \Upsilon^{k+1}. \forall s, s' \in S. \forall q, q' \in Q. \\ & (\lambda^{\mathbb{B}}(s, q, v_1 \dots v_k) \wedge s' = \tau(s, v_{k+1}) \wedge q' \in \delta(q, v_1, l(s))) \\ & \Rightarrow \lambda^{\mathbb{B}}(s', q', v_2 \dots v_{k+1}) \wedge \lambda^{\mathbb{N}}(s, q, v_1 \dots v_k) \sqsupseteq \lambda^{\mathbb{N}}(s', q', v_2 \dots v_{k+1}) \end{aligned}$$

Corollary 4. *Given some $k \geq 0$, if the constraint system is satisfiable for some bound on the size of \mathcal{S}_k then $\mathcal{S}_k \models \mathcal{A}$.*

4 Synthesis with Quantifier Alternations

We now build on the introduced techniques to solve the *synthesis* problem for HyperLTL with quantifier alternation, that is, we search for implementations that satisfy the given properties. In previous work [13], the synthesis problem for $\exists^* \forall^*$ HyperLTL was solved by a reduction to the distributed synthesis problem. We present an alternative synthesis procedure that (1) introduces the necessary concepts for the synthesis of the $\forall^* \exists^*$ fragment and that (2) strictly decomposes the choice of the existential trace quantifier from the implementation.

Fix a formula of the form $\exists^m \forall^n \varphi$. We again reduce the verification problem to the problem of determining whether a run graph is accepting. As the existential quantifiers do not depend on the universal ones, there is no future dependency and thus no need for prophecy variables or bounded lookahead. Formally, \mathcal{S}_{\exists} is a tuple $\langle X, x_0, \mu, l_{\exists} \rangle$ such that X is a set of states, $x_0 \in X$ is the designated initial state, $\mu: X \rightarrow X$ is the transition function, and $l_{\exists}: X \rightarrow \Upsilon^m$ is the labeling function. \mathcal{S}_{\exists} produces infinite sequences of $(\Upsilon^m)^\omega$, without having any knowledge about the behavior of the universally quantified traces. The run graph is then $(\mathcal{S}^m \parallel \mathcal{S}_{\exists}) \times \mathcal{S}^n \times \mathcal{A}_{\varphi}$. The constraint system is built analogously to Sect. 3.2, with the difference that the representation of the system \mathcal{S} is now also uninterpreted. In the resulting SMT constraint system, we have two bounds, one for the size of the implementation \mathcal{S} and one for the size of \mathcal{S}_{\exists} .

Corollary 5. *The bounded synthesis algorithm decides the realizability problem for $\exists^* \forall^1$ HyperLTL and is a semi-decision procedure for $\exists^* \forall^{>1}$ HyperLTL.*

The synthesis problem for formulas in the $\forall^* \exists^*$ HyperLTL fragment uses the same reduction to a constraint system as the strategy synthesis in Sect. 3.2, with the only difference that the transition system \mathcal{S} itself is uninterpreted. In the resulting SMT constraint systems, we have three bounds, the size of the implementation \mathcal{S} , the size of the strategy \mathcal{S}_{\exists} , and the lookahead k .

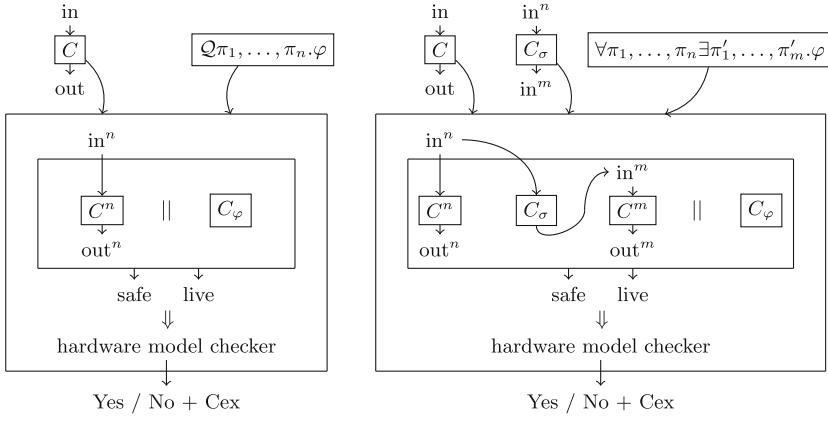


Fig. 1. HyperLTL model checking with MCHYPER

Corollary 6. Given a HyperLTL formula $\forall^n \exists^m \varphi$ where φ is quantifier-free. $\forall^n \exists^m \varphi$ is realizable if the SMT constraint system corresponding to the run graph $\mathcal{S}^n \times (\mathcal{S}^m \parallel \mathcal{S}_\exists) \times \mathcal{A}_\varphi$ is satisfiable for some bounds on \mathcal{S} , \mathcal{S}_\exists , and lookahead k .

5 Implementations and Experimental Evaluation

We have integrated the model checking technique with a manually provided strategy into the HyperLTL hardware model checker MCHYPER¹. For the synthesis of strategies and reactive systems from hyperproperties, we have developed a separate bounded synthesis tool based on SMT-solving. In the following, we describe these implementations and report on experimental results. All experiments ran on a machine with dual-core Core i7, 3.3 GHz, and 16 GB memory.

Hardware Model Checking with Given Strategies. We have extended the model checker MCHYPER [18] from the alternation-free fragment to formulas with one quantifier alternation. The input to MCHYPER is a circuit description as an And-Inverter-Graph in the AIGER format and a HyperLTL formula. Figures 1a and 1 show the model checking process in MCHYPER without and with quantifier alternation, respectively. For formulas with quantifier alternation, the model checker now also accepts a strategy as an additional AIGER circuit C_σ . Based on this strategy, MCHYPER creates a new circuit where only the inputs of the universal system copies are exposed and the inputs of the existential system

¹ Try the online tool interface with the latest version of MCHYPER: <https://www.react.uni-saarland.de/tools/online/MCHyper/>.

Table 1. Experimental results for MCHYPER on the software doping and mutual exclusion benchmarks. All experiments used the IC3 option for ABC. Model and property names correspond to the ones used in [9] and [18].

Model	#Latches	Property	Time[s]
EC 0.05	17	(10.a) + (10.b)	1.8
EC 0.00625	23	(10.a) + (10.b)	53.4
AEC 0.05	19	(\neg 10.a) + (\neg 10.b)	2.8
AEC 0.00625	25	(\neg 10.a) + (\neg 10.b)	160.1
Bakery.a.n.s	47	Sym5	50.6
		Sym6	27.5
Bakery.a.n.s.5proc	90	Sym7	461.3
		Sym8	472.3

copies are determined by the strategy. The new circuit is then model checked as described in [18] with ABC [4].

We evaluate our extension of MCHYPER on formulas with quantifier alternation based on benchmarks from software doping [9] and symmetry in mutual exclusion algorithms [18]. Both considered problems have previously been analyzed with MCHYPER; however, since the properties in both problems require quantifier alternation, we were previously limited to a (manually obtained) approximation of the properties as universal formulas. The correctness of manual approximations is not given but has to be shown separately. By directly model checking the formula with quantifier alternation we know that we are checking the correct formula without needing any additional proof of correctness.

Software Doping. D’Argenio et al. [9] examined a clean and a doped version of an emission control program of a car and used the previous version of MCHYPER to formally verify approximations of these properties. Robust cleanliness is expressed in the one-alternation fragment using two $\forall^2 \exists^1$ HyperLTL formulas (given in Prop. 19 in [9], cf. Sect. 1). In [9], the formulas were strengthened into alternation-free formulas that imply the original properties. Despite the quantifier alternation, Table 1 shows that the new version of MCHYPER verifies the precise formulas in roughly the same time as the alternation-free approximations [9] while giving stronger correctness guarantees.

Symmetry in Mutual Exclusion Protocols. $\forall^* \exists^*$ HyperLTL allows us to specify symmetry for mutual exclusion protocols. In such protocols, we wish to guarantee that every request is eventually answered, and the grants are mutually exclusive. In our experiments, we used an implementation of the Bakery protocol [25]. Table 1 shows the verification results for the precise $\forall^1 \exists^1$ properties. Comparing these results to the performance on the approximations of the symmetry properties [18], we, again, observe that the verification times are similar. However, we gain the additional correctness guarantees as described above.

Strategy and System Synthesis. For the synthesis of strategies for existential quantifiers and for the synthesis of reactive systems from hyperproperties, we have developed a separate bounded synthesis tool based on SMT-solving with z3 [29]. Our evaluation is based on two benchmark families, the *dining cryptographers* problem [5] and a simplified version of the symmetry problem in mutual exclusion protocols discussed previously. The results are shown in Table 2. Obviously, synthesis operates at a vastly smaller scale than model checking with given strategies. In the dining cryptographers example, z3 was unable to find an implementation for the full synthesis problem, but could easily synthesize strategies for the existential trace quantifiers when provided with an implementation. With the progress of constraint solver that employ quantification over Boolean functions [31] we expect scalability improvements of our synthesis approach.

Table 2. Summary of the experimental results on the benchmarks sets described in Sect. 5. When no hyperproperty is given, only the LTL part is used.

Instance	Hyperproperty	$ \mathcal{S} $	$ \mathcal{S}_\exists $	Time [s]
Dining cryptographers	distributed + deniability			TO
	distributed + deniability with given \mathcal{S}	(1)	1	1.2
Mutex	—	2	—	<1
	symmetry	3	1	3.4
Mutex w/o spurious grants	—	3	—	<1
	symmetry	3	1	3.9
	wait-free	3	3	46
	symmetry + wait-free	3	1 + 3840	

6 Conclusions

We have presented model checking and synthesis techniques for hyperliveness properties expressed as HyperLTL formulas with quantifier alternation. The alternation makes it possible to specify hyperproperties such as generalized non-interference, symmetry, and deniability. Our approach is the first method for the synthesis of reactive systems from HyperLTL formulas with quantifier alternation and the first practical method for the verification of such specifications.

The approach is based on a game-theoretic view of existential quantifiers, where the \exists -player reacts to decisions of the \forall -player. The key advantage is that the complementation of the system automaton is avoided (cf. [18]). Instead, a strategy must be found for the \exists -player. Since this can be done either manually or through automatic synthesis, the user of the model checking or synthesis tool has the opportunity to trade some automation for a significant gain in performance.

Acknowledgements. We would like to thank Sebastian Biewer for providing the software doping models and formulas, Marvin Stenger for his advice on our synthesis experiments, and Jana Hofmann for her helpful comments on a draft of this paper.

References

1. Abadi, M., Lamport, L.: The existence of refinement mappings. *Theor. Comput. Sci.* **82**(2), 253–284 (1991). [https://doi.org/10.1016/0304-3975\(91\)90224-P](https://doi.org/10.1016/0304-3975(91)90224-P)
2. Barthe, G., Crespo, J.M., Kunz, C.: Beyond 2-safety: asymmetric product programs for relational program verification. In: Artemov, S., Nerode, A. (eds.) LFCS 2013. LNCS, vol. 7734, pp. 29–43. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35722-0_3
3. Barthe, G., D’Argenio, P.R., Rezk, T.: Secure information flow by self-composition. In: Proceedings of CSFW, pp. 100–114. IEEE Computer Society (2004). <https://doi.org/10.1109/CSFW.2004.17>
4. Brayton, R., Mishchenko, A.: ABC: an academic industrial-strength verification tool. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 24–40. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_5
5. Chaum, D.: Security without identification: transaction systems to make big brother obsolete. *Commun. ACM* **28**(10), 1030–1044 (1985). <https://doi.org/10.1145/4372.4373>
6. Clarkson, M.R., Finkbeiner, B., Koleini, M., Micinski, K.K., Rabe, M.N., Sánchez, C.: Temporal logics for hyperproperties. In: Abadi, M., Kremer, S. (eds.) POST 2014. LNCS, vol. 8414, pp. 265–284. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54792-8_15
7. Clarkson, M.R., Schneider, F.B.: Hyperproperties. *J. Comput. Secur.* **18**(6), 1157–1210 (2010). <https://doi.org/10.3233/JCS-2009-0393>
8. Cook, B., Khlaaf, H., Piterman, N.: On automation of CTL* verification for infinite-state systems. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 13–29. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_2
9. D’Argenio, P.R., Barthe, G., Biewer, S., Finkbeiner, B., Hermanns, H.: Is your software on dope? - formal analysis of surreptitiously “enhanced” programs. In: Yang, H. (ed.) ESOP 2017. LNCS, vol. 10201, pp. 83–110. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54434-1_4
10. D’Souza, D., Holla, R., Raghavendra, K.R., Sprick, B.: Model-checking trace-based information flow properties. *J. Comput. Secur.* **19**(1), 101–138 (2011). <https://doi.org/10.3233/JCS-2010-0400>
11. Finkbeiner, B., Hahn, C.: Deciding hyperproperties. In: Proceedings of CONCUR. LIPIcs, vol. 59, pp. 13:1–13:14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2016). <https://doi.org/10.4230/LIPIcs.CONCUR.2016.13>
12. Finkbeiner, B., Hahn, C., Hans, T.: MGHyper: checking satisfiability of HyperLTL formulas beyond the $\exists^*\forall^*$ fragment. In: Lahiri, S.K., Wang, C. (eds.) ATVA 2018. LNCS, vol. 11138, pp. 521–527. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-01090-4_31
13. Finkbeiner, B., Hahn, C., Lukert, P., Stenger, M., Tentrup, L.: Synthesizing reactive systems from hyperproperties. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 289–306. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_16

14. Finkbeiner, B., Hahn, C., Stenger, M.: EAHyper: satisfiability, implication, and equivalence checking of hyperproperties. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10427, pp. 564–570. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63390-9_29
15. Finkbeiner, B., Hahn, C., Stenger, M., Tentrup, L.: Monitoring hyperproperties. In: Lahiri, S., Reger, G. (eds.) RV 2017. LNCS, vol. 10548, pp. 190–207. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67531-2_12
16. Finkbeiner, B., Hahn, C., Stenger, M., Tentrup, L.: RVHyper: a runtime verification tool for temporal hyperproperties. In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10806, pp. 194–200. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89963-3_11
17. Finkbeiner, B., Hahn, C., Torfah, H.: Model checking quantitative hyperproperties. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 144–163. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_8
18. Finkbeiner, B., Rabe, M.N., Sánchez, C.: Algorithms for model checking Hyper-LTL and HyperCTL*. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 30–48. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_3
19. Finkbeiner, B., Schewe, S.: Uniform distributed synthesis. In: Proceedings of LICS, pp. 321–330. IEEE Computer Society (2005). <https://doi.org/10.1109/LICS.2005.53>
20. Finkbeiner, B., Schewe, S.: Bounded synthesis. STTT **15**(5–6), 519–539 (2013). <https://doi.org/10.1007/s10009-012-0228-z>
21. Goguen, J.A., Meseguer, J.: Security policies and security models. In: Proceedings of S&P, pp. 11–20. IEEE Computer Society (1982). <https://doi.org/10.1109/SP.1982.10014>
22. Hahn, C., Stenger, M., Tentrup, L.: Constraint-based monitoring of hyperproperties. In: Vojnar, T., Zhang, L. (eds.) TACAS 2019. LNCS, vol. 11428, pp. 115–131. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17465-1_7
23. Huisman, M., Worah, P., Sunesen, K.: A temporal logic characterisation of observational determinism. In: Proceedings of CSFW, p. 3. IEEE Computer Society (2006). <https://doi.org/10.1109/CSFW.2006.6>
24. Klein, F., Zimmermann, M.: How much lookahead is needed to win infinite games? In: Halldórsson, M.M., Iwama, K., Kobayashi, N., Speckmann, B. (eds.) ICALP 2015. LNCS, vol. 9135, pp. 452–463. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-47666-6_36
25. Lamport, L.: A new solution of Dijkstra's concurrent programming problem. Commun. ACM **17**(8), 453–455 (1974). <https://doi.org/10.1145/361082.361093>
26. Lynch, N.A., Vaandrager, F.W.: Forward and backward simulations: I. untimed systems. Inf. Comput. **121**(2), 214–233 (1995). <https://doi.org/10.1006/inco.1995.1134>
27. McCullough, D.: Noninterference and the compositability of security properties. In: Proceedings of S&P, pp. 177–186. IEEE Computer Society (1988). <https://doi.org/10.1109/SECPRI.1988.8110>
28. van der Meyden, R., Zhang, C.: Algorithmic verification of noninterference properties. Electr. Notes Theor. Comput. Sci. **168**, 61–75 (2007). <https://doi.org/10.1016/j.entcs.2006.11.002>

29. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
30. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Proceedings of POPL, pp. 179–190. ACM Press (1989). <https://doi.org/10.1145/75277.75293>
31. Tentrup, L., Rabe, M.N.: Clausal abstraction for DQBF. In: Janota, M., Lynce, I. (eds.) SAT 2019. LNCS, vol. 11628, pp. 388–405. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-24258-9_27

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Quantitative Mitigation of Timing Side Channels

Saeid Tizpaz-Niari^(✉), Pavol Černý,
and Ashutosh Trivedi

University of Colorado Boulder, Boulder, USA
Saeid.TizpazNiari@colorado.edu



Abstract. Timing side channels pose a significant threat to the security and privacy of software applications. We propose an approach for *mitigating* this problem by decreasing the strength of the side channels as measured by entropy-based objectives, such as min-guess entropy. Our goal is to minimize the information leaks while guaranteeing a user-specified maximal acceptable performance overhead. We dub the decision version of this problem *Shannon mitigation*, and consider two variants, *deterministic* and *stochastic*. First, we show that the deterministic variant is NP-hard. However, we give a polynomial algorithm that finds an optimal solution from a restricted set. Second, for the stochastic variant, we develop an approach that uses optimization techniques specific to the entropy-based objective used. For instance, for min-guess entropy, we used mixed integer-linear programming. We apply the algorithm to a threat model where the attacker gets to make *functional observations*, that is, where she observes the running time of the program for the same secret value combined with different public input values. Existing mitigation approaches do not give confidentiality or performance guarantees for this threat model. We evaluate our tool SCHMIT on a number of micro-benchmarks and real-world applications with different entropy-based objectives. In contrast to the existing mitigation approaches, we show that in the functional-observation threat model, SCHMIT is scalable and able to maximize confidentiality under the performance overhead bound.

1 Introduction

Information leaks through timing side channels remain a challenging problem [13, 16, 24, 29, 35, 37, 47]. A program leaks secret information through timing side channels if an attacker can deduce secret values (or their properties) by observing response times. We consider the problem of mitigating timing side channels. Unlike elimination techniques [7, 31, 46] that aim to completely remove timing leaks without considering the performance penalty, the goal of mitigation techniques [10, 26, 48] is to weaken the leaks, while keeping the penalty low.

We define the *Shannon mitigation* problem that decides whether there is a mitigation policy to achieve a lower bound on a given security entropy-based

measure while respecting an upper bound on the performance overhead. Consider an example where the program-under-analysis has a secret variable with seven possible values, and has three different timing behaviors, each forming a cluster of secret values. It takes 1 second if the secret value is 1, it takes 5 seconds if the secret is between 2 and 5, and it takes 10 seconds if the secret value is 6 or 7. The *entropy-based measure* quantifies the remaining uncertainty about the secret after timing observations. Min-guess entropy [11, 25, 41] for this program is 1, because if the observed execution time is 1, the attacker guesses the secret in one try. A *mitigation policy* involves merging some timing clusters by introducing delays. A good solution might be to introduce a 9 second delay if the secret is 1, which merges two timing clusters. But, this might be disallowed by the budget on the performance overhead. Therefore, another solution must be found, such as introducing a 4 seconds delay when the secret is one.

We develop two variants of the Shannon mitigation problem: *deterministic* and *stochastic*. The mitigation policy of the deterministic variant requires us to move all secret values associated to an observation to another observation, while the policy of the stochastic variant allows us to move only a portion of secret values in an observation to another one. We show that the deterministic variant of the Shannon mitigation problem is intractable and propose a dynamic programming algorithm to approximate the optimal solution for the problem by searching through a restricted set of solutions. We develop an algorithm that reduces the problem in the stochastic variant to a well-known optimization problem that depends on the entropy-based measure. For instance, with min-guess entropy, the optimization problem is mixed integer-linear programming.

We consider a threat model where an attacker knows the public inputs (known-message attacks [26]), and furthermore, where the public input changes much more often than the secret inputs (for instance, secrets such as bank account numbers do not change often). As a result, for each secret, the attacker observes a timing function of the public inputs. We call this model *functional observations* of timing side channels.

We develop our tool SCHMIT that has three components: side channel discovery [45], search for the mitigation policy, and the policy enforcement. The side channel discovery builds the functional observations [45] and measures the entropy of secret set after the observations. The mitigation policy component includes the implementation of the dynamic programming and optimization algorithms. The enforcement component is a monitoring system that uses the program internals and functional observations to enforce the policy at runtime. To summarize, we make the following contributions:

- We formalize the *Shannon mitigation* problem with two variants and show that the complexity of finding deterministic mitigation policy is NP-hard.
- We describe two algorithms for synthesizing the mitigation policy: one is based on dynamic programming for the deterministic variant, that is in polynomial time and results in an approximate solution, and the other one solves the stochastic variant of the problem with optimization techniques.

- We consider a threat model that results in functional observations. On a set of micro-benchmarks, we show that existing mitigation techniques are not secure and efficient for this threat model.
- We evaluate our approach on five real-world Java applications. We show that SCHMIT is scalable in synthesizing mitigation policy within a few seconds and significantly improves the security (entropy) of the applications.

```
Example(int high, int low) {
    int t_high = high, t_low = low;
    while (t_high > 0) {
        if (t_high % 2 == 1) {
            while (t_low > 0) {
                if (t_low % 2 == 1) {
                    res += compute(t_low,t_high);}
                t_low = t_low >> 1;}}
        t_high = t_high >> 1;}
    return res;}
```

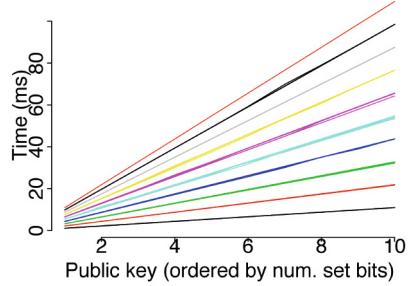


Fig. 1. (a) The example used in Sect. 2. (b) The timing functions for each secret value of the program.

2 Overview

First, we describe the threat model considered in this paper. Second, we describe our approach on a running example. Third, we compare the results of SCHMIT with the existing mitigation techniques [10, 26, 48] and show that SCHMIT achieves the highest entropy (i.e., best mitigation) for all three entropy objectives.

Threat Model. We assume that the attacker has access to the source code and the mitigation model, and she can sample the run-time of the application arbitrarily many times on her own machine. During an attack, she intends to guess a fixed secret of the target machine by observing the mitigated running time. Since we consider the attack models where the attacker knows the public inputs and the secret inputs are less volatile than public inputs, her observations are functional observations, where for each secret value, she learns a function from the public inputs to the running time.

Example 2.1. Consider the program shown in Fig. 1(a). It takes secret and public values as inputs. The running time depends on the number of set bits in both secret and public inputs. We assume that secret and public inputs can be between 1 and 1023. Figure 1(b) shows the running time of different secret values as timing functions, i.e., functions from the public inputs to the running time.

Side channel discovery. One can use existing tools to find the initial functional observations [44, 45]. In Example 2.1, functional observations are $\mathcal{F} = \langle y, 2y, \dots, 10y \rangle$ where y is a variable whose value is the number of set bits in the public input. The corresponding secret classes after this observation is $\mathcal{S}_{\mathcal{F}} = \langle 1_1, 1_2, 1_3, \dots, 1_{10} \rangle$ where 1_n shows a set of secret values that have n set bits. The sizes of classes are $B = \{10, 45, 120, 210, 252, 210, 120, 45, 10, 1\}$. We use L_1 -norm as metric to calculate the distance between the functional observations \mathcal{F} . This distance (penalty) matrix specifies extra performance overhead to move from one functional observation to another. With the assumption of uniform distributions over the secret input, Shannon entropy, guessing entropy, and the min-guessing entropy are 7.3, 90.1, and 1.0, respectively. These entropies are defined in Sect. 3 and measure the remaining entropy of the secret set after the observations. We aim to maximize the entropy measures, while keeping the performance overhead below a threshold, say 60% for this example.

Mitigation with Schmit. We use our tool SCHMIT to mitigate timing leaks of Example 2.1. The mitigation policy for the Shannon entropy objective is shown in Fig. 2(a). The policy results in two classes of observations. The policy requires to move functional observations $\langle y, 2y, \dots, 5y \rangle$ to $\langle 6y \rangle$ and all other observations $\langle 7y, 8y, 9y \rangle$ to $\langle 10y \rangle$. To enforce this policy, we use a monitoring system at runtime. The monitoring system uses a decision tree model of the initial functional observations. The decision tree model characterizes each functional observation with associated program internals such as method calls or basic block invocations [43, 44]. The decision tree model for the Example 2.1 is shown in Fig. 2(b). The monitoring system records program internals and matches it with the decision tree model to detect the current functional observation. Then, it adds delays, if necessary, to the execution time in order to enforce the mitigation policy. With this method, the mitigated functional observation is $\mathcal{G} = \langle 6y, 10y \rangle$ and the secret

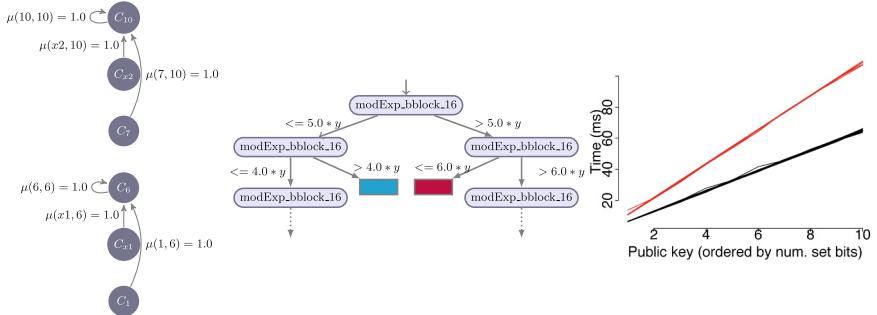


Fig. 2. (a) Mitigation policy calculation with deterministic algorithm (left). The observations $x1$ and $x2$ stands for all observations from C_2-C_5 and from C_8-C_9 , resp.; (b) Leaned discriminant decision tree (center): it characterizes the functional clusters of Fig. 1(b) with internals of the program in Fig. 1(a); and (c) observations (right) after the mitigation by SCHMIT results in two classes of observations.

class is $\mathcal{S}_G = \langle \{1_1, 1_2, 1_3, 1_4, 1_5, 1_6\}, \{1_7, 1_8, 1_9, 1_{10}\} \rangle$ as shown in Fig. 2 (c). The performance overhead of this mitigation is 43.1%. The Shannon, guessing, and min-guess entropies have improved to 9.7, 459.6, and 193.5, respectively.

Comparison with state of the art. We compare our mitigation results to black-box mitigation scheme [10] and bucketing [26]. *Black-box double scheme technique.* We use the double scheme technique [10] to mitigate the leaks of Example 2.1. This mitigation uses a prediction model to release events at scheduled times. Let us consider the prediction for releasing the event i at N -th epoch with $S(N, i) = \max(\text{inp}_i, S(N, i-1)) + p(N)$, where inp_i is the time arrival of the i -th request, $S(N, i-1)$ is the prediction for the request $i-1$, and $p(N) = 2^{N-1}$ models the basis for the prediction scheme at N -th epoch. We assume that the requests are the same type and the sequence of public input requests for each secret are received in the beginning of epoch $N = 1$. Figure 3(a) shows the functional observations after applying the predictive mitigation. With this mitigation, the classes of observations are $\mathcal{S}_G = \langle \{1_1, \{1_2, 1_3\}, \{1_4, 1_5, 1_6, 1_7\}, \{1_8, 1_9, 1_{10}\} \rangle$. The number of classes of observations is reduced from 10 to 4. The performance overhead is 39.9%. The Shannon, guessing, and min-guess entropies have increased to 9.00, 321.5, and 5.5, respectively. *Bucketing.* We consider the mitigation approach with buckets [26]. For Example 2.1, if the attacker does not know the public input (unknown-message attacks [26]), the observations are $\{1.1, 2.1, 3.3, \dots, 9.9, 10.9, \dots, 109.5\}$ as shown in Fig. 3(b). We apply the bucketing algorithm in [26] for these observations, and it finds two buckets $\{37.5, 109.5\}$ shown with the red lines in Fig. 3(b). The bucketing mitigation requires to move the observations to the closest bucket. Without functional observations, there are 2 classes of observations. However, with functional observations, there are more than 2 observations. Figure 3(c) shows how the pattern of observations are leaking through functional side channels. There are 7 classes of observations: $\mathcal{S}_G = \langle \{1_1, 1_2, 1_3\}, \{1_4\}, \{1_5\}, \{1_6\}, \{1_7\}, \{1_8\}, \{1_9\}, \{1_{10}\} \rangle$. The Shannon, guessing, and min-guess entropies are 7.63, 102.3, and 1.0, respectively.

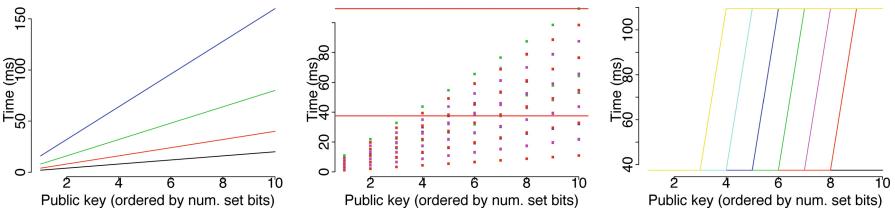


Fig. 3. (a) The execution time after mitigation using the double scheme technique [10]. There are four classes of functional observations after the mitigation. (b) Mitigation with bucketing [26]. All observations require to move to the closest red line. (c) Functional observations distinguish 7 classes of observations after mitigating with bucketing.

Overall, SCHMIT achieves the higher entropy measures for all three objectives under the performance overhead of 60%.

3 Preliminaries

For a finite set Q , we use $|Q|$ for its cardinality. A *discrete probability distribution*, or just distribution, over a set Q is a function $d : Q \rightarrow [0, 1]$ such that $\sum_{q \in Q} d(q) = 1$. Let $\mathcal{D}(Q)$ denote the set of all discrete distributions over Q . We say a distribution $d \in \mathcal{D}(Q)$ is a *point distribution* if $d(q)=1$ for a $q \in Q$. Similarly, a distribution $d \in \mathcal{D}(Q)$ is *uniform* if $d(q)=1/|Q|$ for all $q \in Q$.

Definition 1 (Timing Model). *The timing model of a program \mathcal{P} is a tuple $\llbracket \mathcal{P} \rrbracket = (X, Y, \mathcal{S}, \delta)$ where $X = \{x_1, \dots, x_n\}$ is the set of secret-input variables, $Y = \{y_1, \dots, y_m\}$ is the set of public-input variables, $\mathcal{S} \subseteq \mathbb{R}^n$ is a finite set of secret-inputs, and $\delta : \mathbb{R}^n \times \mathbb{R}^m \rightarrow \mathbb{R}_{\geq 0}$ is the execution-time function of the program over the secret and public inputs.*

We assume that the adversary knows the program and wishes to learn the value of the secret input. To do so, for some fixed secret value $s \in \mathcal{S}$, the adversary can invoke the program to estimate (to an arbitrary precision) the execution time of the program. If the set of public inputs is empty, i.e. $m = 0$, the adversary can only make *scalar observations* of the execution time corresponding to a secret value. In the more general setting, however, the adversary can arrange his observations in a functional form by estimating an approximation of the *timing function* $\delta(s) : \mathbb{R}^m \rightarrow \mathbb{R}_{\geq 0}$ of the program.

A *functional observation* of the program \mathcal{P} for a secret input $s \in \mathcal{S}$ is the function $\delta(s) : \mathbb{R}^m \rightarrow \mathbb{R}_{\geq 0}$ defined as $\mathbf{y} \in \mathbb{R}^m \mapsto \delta(s, \mathbf{y})$. Let $\mathcal{F} \subseteq [\mathbb{R}^m \rightarrow \mathbb{R}_{\geq 0}]$ be the finite set of all functional observations of the program \mathcal{P} . We define an order \prec over the functional observations \mathcal{F} : for $f, g \in \mathcal{F}$ we say that $f \prec g$ if $f(y) \leq g(y)$ for all $y \in \mathbb{R}^m$.

The set \mathcal{F} characterizes an equivalence relation $\equiv_{\mathcal{F}}$, namely secrets with equivalent functional observations, over the set \mathcal{S} , defined as following: $s \equiv_{\mathcal{F}} s'$ if there is an $f \in \mathcal{F}$ such that $\delta(s) = \delta(s') = f$. Let $\mathcal{S}_{\mathcal{F}} = \langle S_1, S_2, \dots, S_k \rangle$ be the quotient space of \mathcal{S} characterized by the observations $\mathcal{F} = \langle f_1, f_2, \dots, f_k \rangle$. We write \mathcal{S}_f for the secret set $S \in \mathcal{S}_{\mathcal{F}}$ corresponding to the observations $f \in \mathcal{F}$. Let $\mathcal{B} = \langle B_1, B_2, \dots, B_k \rangle$ be the size of observational equivalence class in $\mathcal{S}_{\mathcal{F}}$, i.e. $B_i = |\mathcal{S}_{f_i}|$ for $f_i \in \mathcal{F}$ and let $B = |\mathcal{S}| = \sum_{i=1}^k B_i$.

Shannon entropy, guessing entropy, and min-guess entropy are three prevalent information metrics to quantify information leaks in programs. Köpf and Basin [25] characterize expressions for various information-theoretic measures on information leaks when there is a uniform distribution on \mathcal{S} given below.

Proposition 1 (Köpf and Basin [25]). *Let $\mathcal{F} = \langle f_1, \dots, f_k \rangle$ be a set of observations and let \mathcal{S} be the set of secret values. Let $\mathcal{B} = \langle B_1, \dots, B_k \rangle$ be the*

corresponding size of secret set in each class of observation and $B = \sum_{i=1}^k B_i$. Assuming a uniform distribution on \mathcal{S} , entropies can be characterized as:

1. **Shannon Entropy:** $SE(\mathcal{S}|\mathcal{F}) \stackrel{\text{def}}{=} (\frac{1}{B}) \sum_{1 \leq i \leq k} B_i \log_2(B_i)$,
2. **Guessing Entropy:** $GE(\mathcal{S}|\mathcal{F}) \stackrel{\text{def}}{=} (\frac{1}{2B}) \sum_{1 \leq i \leq k} B_i^2 + \frac{1}{2}$, and
3. **Min-Guess Entropy:** $mGE(\mathcal{S}|\mathcal{F}) \stackrel{\text{def}}{=} \min_{1 \leq i \leq k} \{(B_i + 1)/2\}$.

4 Shannon Mitigation Problem

Our goal is to mitigate the information leakage due to the timing side channels by adding synthetic delays to the program. An aggressive, but commonly-used, mitigation strategy aims to eliminate the side channels by adding delays such that every secret value yields a common functional observation. However, this strategy may often be impractical as it may result in unacceptable performance degradations of the response time. Assuming a well-known penalty function associated with the performance degradation, we study the problem of maximizing entropy while respecting a bound on the performance degradation. We dub the decision version of this problem Shannon mitigation.

Adding synthetic delays to execution-time of the program, so as to mask the side-channel, can give rise to new functional observations that correspond to upper-envelopes of various combinations of original observations. Let $\mathcal{F} = \langle f_1, f_2, \dots, f_k \rangle$ be the set of functional observations. For $I \subseteq 1, 2, \dots, k$, let $f_I = \mathbf{y} \in \mathbb{R}^m \mapsto \sup_{i \in I} f_i(\mathbf{y})$ be the functional observation corresponding to upper-envelope of the functional observations in the set I . Let $\mathcal{G}(\mathcal{F}) = \{f_I : I \neq \emptyset \subseteq \{1, 2, \dots, k\}\}$ be the set of all possible functional observations resulting from the upper-envelope calculations. To change the observation of a secret value with functional observation f_i to a new observation f_I (we assume that $i \in I$), we need to add delay function $f_I^i : \mathbf{y} \in \mathbb{R}^m \mapsto f_I(\mathbf{y}) - f_i(\mathbf{y})$.

Mitigation Policies. Let $\mathcal{G} \subseteq \mathcal{G}(\mathcal{F})$ be a set of admissible post-mitigation observations. A *mitigation policy* is a function $\mu : \mathcal{F} \rightarrow \mathcal{D}(\mathcal{G})$ that for each secret $s \in \mathcal{S}_f$ suggests the probability distribution $\mu(f)$ over the functional observations. We say that a mitigation policy is *deterministic* if for all $f \in \mathcal{F}$ we have that $\mu(f)$ is a point distribution. Abusing notations, we represent a deterministic mitigation policy as a function $\mu : \mathcal{F} \rightarrow \mathcal{G}$. The semantics of a mitigation policy recommends to a program analyst a probability $\mu(f)(g)$ to elevate a secret input $s \in \mathcal{S}_f$ from the observational class f to the class $g \in \mathcal{G}$ by adding $\max\{0, g(p) - f(p)\}$ units delay to the corresponding execution-time $\delta(s, p)$ for all $p \in Y$. We assume that the mitigation policies respect the order, i.e. for every mitigation policy μ and for all $f \in \mathcal{F}$ and $g \in \mathcal{G}$, we have that $\mu(f)(g) > 0$ implies that $f \prec g$. Let $M_{(\mathcal{F} \rightarrow \mathcal{G})}$ be the set of mitigation policies from the set of observational clusters \mathcal{F} into the clusters \mathcal{G} .

For the functional observations $\mathcal{F} = \langle f_1, \dots, f_k \rangle$ and a mitigation policy $\mu \in M_{(\mathcal{F} \rightarrow \mathcal{G})}$, the resulting observation set $\mathcal{F}[\mu] \subseteq \mathcal{G}$ is defined as:

$$\mathcal{F}[\mu] = \{g \in \mathcal{G} : \text{ there exists } f \in \mathcal{F} \text{ such that } \mu(f)(g) > 0\}.$$

Since the mitigation policy is stochastic, we use average sizes of resulting observations to represent fitness of a mitigation policy. For $\mathcal{F}[\mu] = \langle g_1, g_2, \dots, g_\ell \rangle$, we define their expected class sizes $\mathcal{B}_\mu = \langle C_1, C_2, \dots, C_\ell \rangle$ as $C_i = \sum_{j=1}^i \mu(f_j)(f_i) \cdot B_j$ (observe that $\sum_{i=1}^\ell C_i = B$). Assuming a uniform distribution on \mathcal{S} , various entropies for the expected class size after applying a policy $\mu \in M_{(\mathcal{F} \rightarrow \mathcal{G})}$ can be characterized by the following expressions:

1. **Shannon Entropy:** $\text{SE}(\mathcal{S}|\mathcal{F}, \mu) \stackrel{\text{def}}{=} (\frac{1}{B}) \sum_{1 \leq i \leq \ell} C_i \log_2(C_i)$,
2. **Guessing Entropy:** $\text{GE}(\mathcal{S}|\mathcal{F}, \mu) \stackrel{\text{def}}{=} (\frac{1}{2B}) \sum_{1 \leq i \leq \ell} C_i^2 + \frac{1}{2}$, and
3. **Min-Guess Entropy:** $\text{mGE}(\mathcal{S}|\mathcal{F}, \mu) \stackrel{\text{def}}{=} \min_{1 \leq i \leq \ell} \{(C_i + 1)/2\}$.

We note that the above definitions do not represent the expected entropies, but rather entropies corresponding to the expected cluster sizes. However, the three quantities provide bounds on the expected entropies after applying μ . Since Shannon and Min-Guess entropies are concave functions, from Jensen's inequality, we get that $\text{SE}(\mathcal{S}|\mathcal{F}, \mu)$ and $\text{mGE}(\mathcal{S}|\mathcal{F}, \mu)$ are upper bounds on expected Shannon and Min-Guess entropies. Similarly, $\text{GE}(\mathcal{S}|\mathcal{F}, \mu)$, being a convex function, give a lower bound on expected guessing entropy.

We are interested in maximizing the entropy while respecting constraints on the overall performance of the system. We formalize the notion of performance by introducing performance penalties: there is a function $\pi : \mathcal{F} \times \mathcal{G} \rightarrow \mathbb{R}_{\geq 0}$ such that elevating from the observation $f \in \mathcal{F}$ to the functional observation $g \in \mathcal{G}$ adds an extra $\pi(f, g)$ performance overheads to the program. The expected performance penalty associated with a policy μ , $\pi(\mu)$, is defined as the probabilistically weighted sum of the penalties, i.e. $\sum_{f \in \mathcal{F}, g \in \mathcal{G}: f \prec g} |\mathcal{S}_f| \cdot \mu(f)(g) \cdot \pi(f, g)$. Now, we introduce our key decision problem.

Definition 2 (Shannon Mitigation). *Given a set of functional observations $\mathcal{F} = \langle f_1, \dots, f_k \rangle$, a set of admissible post-mitigation observations $\mathcal{G} \subseteq \mathcal{G}(\mathcal{F})$, set of secrets \mathcal{S} , a penalty function $\pi : \mathcal{F} \times \mathcal{G} \rightarrow \mathbb{R}_{\geq 0}$, a performance penalty upper bound $\Delta \in \mathbb{R}_{\geq 0}$, and an entropy lower-bound $E \in \mathbb{R}_{\geq 0}$, the Shannon mitigation problem $\text{SHAN}_{\mathcal{E}}(\mathcal{F}, \mathcal{G}, \mathcal{S}, \pi, E, \Delta)$, for a given entropy measure $\mathcal{E} \in \{\text{SE}, \text{GE}, \text{mGE}\}$, is to decide whether there exists a mitigation policy $\mu \in M_{(\mathcal{F} \rightarrow \mathcal{G})}$ such that $\mathcal{E}(\mathcal{S}|\mathcal{F}, \mu) \geq E$ and $\pi(\mu) \leq \Delta$. We define the deterministic Shannon mitigation variant where the goal is to find a deterministic such policy.*

5 Algorithms for Shannon Mitigation Problem

5.1 Deterministic Shannon Mitigation

We first establish the intractability of the deterministic variant.

Theorem 1. *Deterministic Shannon mitigation problem is NP-complete.*

Proof. It is easy to see that the deterministic Shannon mitigation problem is in NP: one can guess a certificate as a deterministic mitigation policy $\mu \in M_{(\mathcal{F} \rightarrow \mathcal{G})}$

and can verify in polynomial time that it satisfies the entropy and overhead constraints. Next, we sketch the hardness proof for the min-guess entropy measure by providing a reduction from the *two-way partitioning* problem [28]. For the Shannon entropy and guess entropy measures, a reduction can be established from the Shannon capacity problem [18] and the Euclidean sum-of-squares clustering problem [8], respectively.

Given a set $A = \{a_1, a_2, \dots, a_k\}$ of integer values, the two-way partitioning problem is to decide whether there is a partition $A_1 \uplus A_2 = A$ into two sets A_1 and A_2 with equal sums, i.e. $\sum_{a \in A_1} a = \sum_{a \in A_2} a$. W.l.o.g assume that $a_i \leq a_j$ for $i \leq j$. We reduce this problem to a deterministic Shannon mitigation problem $\text{SHAN}_{\text{mGE}}(\mathcal{F}_A, \mathcal{G}_A, \mathcal{S}_A, \pi_A, E_A, \Delta_A)$ with k clusters $\mathcal{F}_A = \mathcal{G}_A = \langle f_1, f_2, \dots, f_k \rangle$ with the secret set $\mathcal{S}_A = \langle S_1, S_2, \dots, S_k \rangle$ such that $|S_i| = a_i$. If $\sum_{1 \leq i \leq k} a_i$ is odd then the solution to the two-way partitioning instance is trivially no. Otherwise, let $E_A = (1/2) \sum_{1 \leq i \leq k} a_i$. Notice that any deterministic mitigation strategy that achieves min-guess entropy larger than or equal to E_A must have at most two clusters. On the other hand, the best min-guess entropy value can be achieved by having just a single cluster. To avoid this and force getting two clusters corresponding to the two partitions of a solution to the two-way partitions problem instance A , we introduce performance penalties such that merging more than $k - 2$ clusters is disallowed by keeping performance penalty $\pi_A(f, g) = 1$ and performance overhead $\Delta_A = k - 2$. It is straightforward to verify that an instance of the resulting min-guess entropy problem has a yes answer if and only if the two-way partitioning instance does. \square

Since the deterministic Shannon mitigation problem is intractable, we design an approximate solution for the problem. Note that the problem is hard even if we only use existing functional observations for mitigation, i.e., $\mathcal{G} = \mathcal{F}$. Therefore, we consider this case for the approximate solution. Furthermore, we assume the following *sequential dominance* restriction on a deterministic policy μ : for $f, g \in \mathcal{F}$ if $f \prec g$ then either $\mu(f) \prec g$ or $\mu(f) = \mu(g)$. In other words, for any given $f \prec g$, f can not be moved to a higher cluster than g without having g be moved to that cluster. For example, Fig. 4(a) shows Shannon mitigation problem with four functional observations and all possible mitigation policies (we represent $\mu(f_i)(f_j)$ with $\mu(i, j)$). Figure 4(b) satisfies the sequential dominance restriction, while Fig. 4(c) does not.

The search for the deterministic policies satisfying the sequential dominance restriction can be performed efficiently using dynamic programming by effective use of intermediate results' memorizations.

Algorithm (1) provides a pseudocode for the dynamic programming solution to find a deterministic mitigation policy satisfying the sequential dominance. The key idea is to start with considering policies that produce a single cluster for subclasses P_i of the problem with the observation from $\langle f_1, \dots, f_i \rangle$, and then compute policies producing one additional cluster in each step by utilizing the previously computed sub-problems and keeping track of the performance penalties. The algorithm terminates as soon as the solution of the current step respects the performance bound. The complexity of the algorithm is $O(k^3)$.

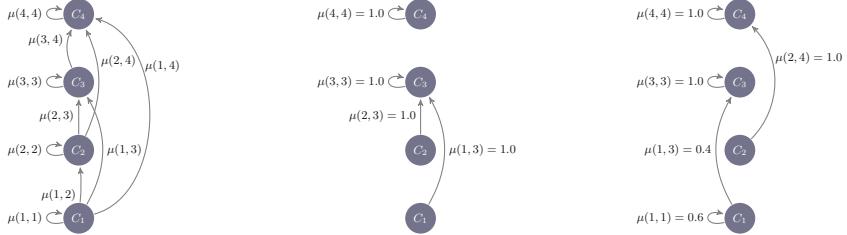


Fig. 4. (a). Example of Shannon mitigation problem with all possible mitigation policies for 4 classes of observations. (b,c) Two examples of the mitigation policies that results in 2 and 3 classes of observations.

5.2 Stochastic Shannon Mitigation Algorithm

Next, we solve the (stochastic) Shannon mitigation problem by posing it as an optimization problem. Consider the stochastic Shannon mitigation problem $\text{SHAN}_{\mathcal{E}} (\mathcal{F}, \mathcal{G} = \mathcal{F}, \mathcal{S}_{\mathcal{F}}, \pi, E, \Delta)$ with a stochastic policy $\mu : \mathcal{F} \rightarrow \mathcal{D}(\mathcal{G})$ and

Algorithm 1. APPROXIMATE DETERMINISTIC SHANNON MITIGATION

Input: The Shannon entropy problem $\text{SHAN}_{MGE} (\mathcal{F}, \mathcal{G} = \mathcal{F}, \mathcal{S}_{\mathcal{F}}, \pi, E, \Delta)$
Output: The entropy table (T).

```

1  for  $i = 1$  to  $k$  do
2     $T(i, 1) = \mathcal{E}(\bigcup_{j=1}^i S_j)$ 
3    if  $\sum_{1 \leq j \leq i} \pi(j, i)(B_j/B) \leq \Delta$  then  $\Pi(i, 1) = \sum_{1 \leq j \leq i} \pi(j, i)(B_j/B)$ 
4    else  $\Pi(i, 1) = \infty$ 
5  if  $\Pi(k, 1) < \infty$  then return  $T$ ;
6  for  $r = 2$  to  $k$  do
7    for  $i = 1$  to  $k$  do
8       $\Omega(i, r) = \{j : 1 \leq j < i \text{ and } \Pi(j, r-1) + \sum_{j < q \leq i} \pi(q, i)(B_q/B) \leq \Delta\}$ 
9      if  $\Omega \neq \emptyset$  then  $T(i, r) = \max_{j \in \Omega(i, r)} \left( \min(T(j, r-1), \mathcal{E}(\bigcup_{q=j+1}^i S_q)) \right)$ 
10     else  $T(i, r) = -\infty$ 
11     Let  $j$  be the index that maximizes  $T(i, r)$ 
12     if  $\Omega \neq \emptyset$  then  $\Pi(i, r) = (\Pi(j, r-1) + \sum_{j < q \leq i} \pi(q, i)(B_q/B))$ 
13     else  $\Pi(i, r) = \infty$ 
14   if  $\Pi(k, r) < \infty$  then return  $T$ ;
15 return  $T$ ;

```

$S_{\mathcal{F}} = \langle S_1, S_2, \dots, S_k \rangle$. The following program characterizes the optimization problem that solves the Shannon mitigation problem with stochastic policy.

Maximize \mathcal{E} , subject to:

1. $0 \leq \mu(f_i)(f_j) \leq 1$ for $1 \leq i \leq j \leq k$
2. $\sum_{i \leq j \leq k} \mu(f_i)(f_j) = 1$ for all $1 \leq i \leq k$.
3. $\sum_{i=1}^k \sum_{j=i}^k |S_i| \cdot \mu(f_i)(f_j) \cdot \pi(f_i, f_j) \leq \Delta$.
4. $C_j = \sum_{i=1}^j |S_i| \cdot \mu(f_i)(f_j)$ for $1 \leq j \leq k$.

Here, the objective function \mathcal{E} is one of the following functions:

1. **Guessing Entropy** $\mathcal{E}_{GE} = \sum_{j=1}^k C_j^2$
2. **Min-Guess Entropy** $\mathcal{E}_{MGE} = \min_{1 \leq j \leq k} \{C_j \mid C_j > 0\}$
3. **Shannon Entropy** $\mathcal{E}_{SE} = \sum_{j=1}^k C_j \cdot \log_2(C_j)$

The linear constraints for the problem are defined as the following. The condition (1) and (2) express that μ provides a probability distributions, condition (3) provides restrictions regarding the performance constraint, and the condition (4) is the entropy specific constraint. The objective function of the optimization problem is defined based on the entropy criteria from \mathcal{E} . For the simplicity, we omit the constant terms from the objective function definitions. For the guessing entropy, the problem is an instance of linearly constrained quadratic optimization problem [33]. The problem with Shannon entropy is a non-linear optimization problem [12]. Finally, the optimization problem with min-guess entropy is an instance of mixed integer programming [32]. We evaluate the scalability of these solvers empirically in Sect. 6 and leave the exact complexity as an open problem. We show that the min-guess entropy objective function can be efficiently solved with the branch and bound algorithms [36]. Figure 4(b,c) show two instantiations of the mitigation policies that are possible for the stochastic mitigation.

6 Implementation Details

A. Environmental Setups. All timing measurements are conducted on an Intel NUC5i5RYH. We switch off JIT Compilation and run each experiment multiple times and use the mean running time. This helps to reduce the effects of environmental factors such as the Garbage Collections. All other analyses are conducted on an Intel i5-2.7 GHz machine.

B. Implementation of Side Channel Discovery. We use the technique presented in [45] for the side channel discovery. The technique applies the functional

data analysis [38] to create B-spline basis and fit functions to the vector of timing observations for each secret value. Then, the technique applies the functional data clustering [21] to obtain K classes of observations. We use the number of secret values in a cluster as the class size metric and the L_1 distance norm between the clusters as the penalty function.

C. Implementation of Mitigation Policy Algorithms. For the stochastic optimization, we encode the Shannon entropy and guessing entropy with linear constraints in Scipy [22]. Since the objective functions are non-linear (for the Shannon entropy) and quadratic (for the guessing entropy), Scipy uses sequential least square programming (SLSQP) [34] to maximize the objectives. For the stochastic optimization with the min-guess entropy, we encode the problem in Gurobi [19] as a mixed-integer programming (MIP) problem [32]. Gurobi solves the problem efficiently with branch-and-bound algorithms [1]. We use Java to implement the dynamic programming.

D. Implementation of Enforcement. The enforcement of mitigation policy is implemented in two steps. *First*, we use the initial timing functions and characterize them with program internal properties such as basic block calls. To do so, we use the decision tree learning approach presented in [45]. The decision tree model characterizes each functional observations with properties of program internals. *Second*, given the policy of mitigation, we enforce the mitigation policy with a monitoring system implemented on top of the Javassist [15] library. The monitoring system uses the decision tree model and matches the properties enabled during an execution with the tree model (detection of the current cluster). Then, it adds extra delays, based on the mitigation policy, to the current execution-time and enforces the mitigation policy. Note that the dynamic monitoring can result in a few micro-second delays. For the programs with timing differences in the order of micro-seconds, we transform source code using the decision tree model. The transformation requires manual efforts to modify and compile the new program. But, it adds negligible delays.

E. Micro-benchmark Results. Our goal is to compare different mitigation methods in terms of their security and performance. We examine the computation time of our tool SCHMIT in calculating the mitigation policies. See appendix for the relationships between performance bounds and entropy measures.

Applications: Mod_Exp applications [30] are instances of square-and-multiply modular exponentiation ($R = y^k \bmod n$) used for secret key operations in RSA [39]. Branch_and_Loop series consist of 6 applications where each application has conditions over secret values and runs a linear loop over the public values. The running time of the applications depend on the slope of the linear loops determined by the secret input.

Computation time comparisons: Fig. 5 shows the computation time for Branch_and_Loop applications (the applications are ordered in x-axis based on the discovered number of observational classes). For the min-guess entropy, we observe that both stochastic and dynamic programming approaches are efficient and fast as shown in Fig. 5(a). For the Shannon and guessing entropies,

Table 1. Micro-benchmark results. M.E and B.L stand for Mod_Exp and Branch_and_Loop applications. Legend: #S: no. of secret values, #P: no. of public values, Δ : Upper bound over performance penalty, ϵ : clustering parameter, #K: classes of observations before mitigation, #K_X: classes of observations after mitigation with X technique, mGE: Min-guess entropy before mitigation, mGE_X: Min-guess entropy after mitigation with X, O_X: Performance overhead added after mitigation with X.

App(s)	Initial Characteristics						Double Scheme			Bucketting			SCHMIT (Determ.)			SCHMIT (Stoch.)		
	#S	#P	Δ	ϵ	#K	mGE	#K _{DS}	mGE _{DS}	O _{DS} (%)	#K _B	mGE _B	O _B (%)	K _D	#mGE _D	O _D (%)	#K _S	mGE _S	O _S (%)
M.E.1	32	0.5	1.0	1	16.5	1	16.5	0.0	1	16.5	0.0	1	16.5	0.0	1	16.5	0.0	
M.E.2	64	0.5	1.0	2	16.5	1	32.5	5,221	1	32.5	27.6	1	32.5	21.4	1	32.5	21.4	
M.E.3	128	0.5	2.0	2	32.5	1	64.5	5,407	1	64.5	33.9	1	64.5	22.7	1	64.5	22.7	
M.E.4	256	0.5	2.0	4	10.5	1	128.5	6,679	1	128.5	30.7	1	128.5	28.3	1	128.5	28.3	
M.E.5	512	0.5	5.0	23	1.0	1	256.5	7,294	2	128.5	50.0	1	256.5	31.0	1	253.0	30.3	
M.E.6	1,024	0.5	8.0	40	1.0	1	512.5	7,822	20	1.0	34.5	2	27.5	46.7	5	85.5	50.0	
B.L.1	25	0.5	10.0	4	3.0	3	3.0	73.0	3	3.0	17.5	2	5.5	26.1	2	6.5	34.9	
B.L.2	50	0.5	10.0	8	3.0	4	3.0	61.3	5	3.0	21.9	2	10.5	45.3	2	13.0	45.3	
B.L.3	100	0.5	20.0	16	3.0	4	8.0	42.4	8	3.0	33.4	2	20.5	48.3	2	21.5	50	
B.L.4	200	0.5	20.0	32	3.0	6	3.0	36.9	16	3.0	28.7	2	48.0	48.7	2	50.5	49.7	
B.L.5	400	0.5	20.0	64	3.0	8	3.0	35.4	32	3.0	27.2	3	65.5	32.0	2	100.5	50.0	
B.L.6	800	0.5	20.0	125	3.0	12	8.0	37.8	29	3.0	52.5	3	133.0	34.6	2	200.5	49.6	

the dynamic programming is scalable, while the stochastic mitigation is computationally expensive beyond 60 classes of observations as shown in Fig. 5(b,c).

Mitigation Algorithm Comparisons: Table 1 shows micro-benchmark results that compare the four mitigation algorithms with the two program series. Double scheme mitigation technique [10] does not provide guarantees on the performance overhead, and we can see that it is increased by more than 75 times for mod_exp_6. Double scheme method reduces the number of classes of observations. However, we observe that this mitigation has difficulty improving the min-guess entropy. Second, Bucketing algorithm [26] can guarantee the performance overhead, but it is not an effective method to improve the security of functional observations, see the examples mod_exp_6 and Branch_and_Loop_6. Third, in the algorithms, SCHMIT guarantees the performance to be below a certain bound, while it results in the highest entropy values. In most cases, the stochastic optimization technique achieves the highest min-entropy value. Here, we show the results with min-guess entropy measure. Also, we have strong evidences to show that SCHMIT achieves higher Shannon and guessing entropies. For example, in B_L_5, the initial Shannon entropy has improved from 2.72 to 6.62, 4.1, 7.56, and 7.28 for the double scheme, the bucketing, the stochastic, and the deterministic algorithms, respectively.

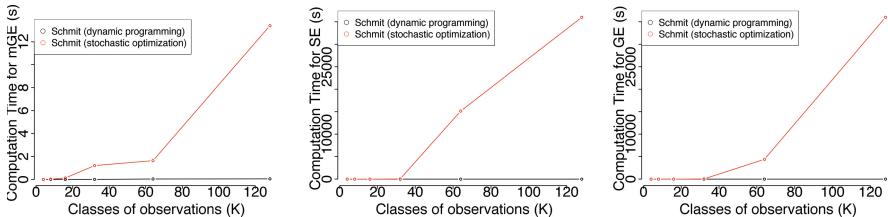


Fig. 5. Computation time for synthesizing mitigation policy over Branch_and_Loop applications. Computation time for min-guess entropy (a) takes only few seconds. Computation time for the Shannon entropy (b) and guessing entropy (c) are expensive using Stochastic optimization. We set time-out to be 10 hours.

7 Case Study

Research Question. Does SCHMIT scale well and improve the security of applications (entropy measures) within the given performance bounds?

Methodology. We use the deterministic and stochastic algorithms for mitigating the leaks. We show our results for the min-guess entropy, but other entropy measures can be applied as well. Since the task is to mitigate existing leakages, we assume that the secret and public inputs are given.

Objects of Study. We consider four real-world applications:

In the inset table, we show the basic characteristics of these benchmarks.

Application	Num methods	Num secret	Num public	ϵ	Initial clusters	Initial Min-guess
GabFeed	573	1,105	65	6.50	34	1.0
Jetty	63	800	635	0.1	20	4.5
Java Verbal Expressions	61	2,000	10	0.02	9	50.5
Password Checker	6	20	2,620	0.05	6	1.0

GabFeed is a chat server with 573 methods [4]. There is a side channel in the authentication part of the application where the application takes users' public keys and its own private key, and generating a common key [14]. The vulnerability leaks the number of set bits in the secret key. Initial functional observations are shown in Fig. 6a. There are 34 clusters and min-guess entropy is 1. We aim to maximize the min-guess entropy under the performance overhead of 50%.

Jetty. We mitigate the side channels in `util.security` package of Eclipse Jetty web server. The package has `Credential` class which had a timing side channel. This vulnerability was analyzed in [14] and fixed initially in [6]. Then, the developers noticed that the implementation in [6] can still leak information and fixed this issue with a new implementation in [5]. However, this new implementation is still leaking information [45]. We apply SCHMIT to mitigate this timing side channels. Initial functional observations is shown in Fig. 6d. There are 20 classes of observations and the initial min-guess entropy is 4.5. We aim to maximize the min-guess entropy under the performance overhead of 50%.

Java Verbal Expressions is a library with 61 methods that construct regular expressions [2]. There is a timing side channel in the library similar to password comparison vulnerability [3] if the library has secret inputs. In this case, starting from the initial character of a candidate expression, if the character matches with the regular expression, it slightly takes more time to respond the request than otherwise. This vulnerability can leak all the regular expressions. We consider regular expressions to have a maximum size of 9. There are 9 classes of observations and the initial min-guess entropy is 50.5. We aim to maximize the min-guess entropy under the performance overhead of 50%.

Password Checker. We consider the password matching example from loginBad program [9]. The password stored in the server is secret, and the user's guess is a public input. We consider 20 secret (lengths at most 6) and 2,620 public inputs. There are 6 different clusters, and the initial min-guess entropy is 1.

Findings for GabFeed. With the stochastic algorithm, SCHMIT calculates the mitigation policy that results in 4 clusters. This policy improves the min-guess entropy from 1 to 138.5 and adds an overhead of 42.8%. With deterministic algorithm, SCHMIT returns 3 clusters. The performance overhead is 49.7% and the min-guess entropy improves from 1 to 106. The user chooses the deterministic policy and enforces the mitigation. We apply CART decision tree learning and characterizes the classes of observations with GabFeed method calls as shown in

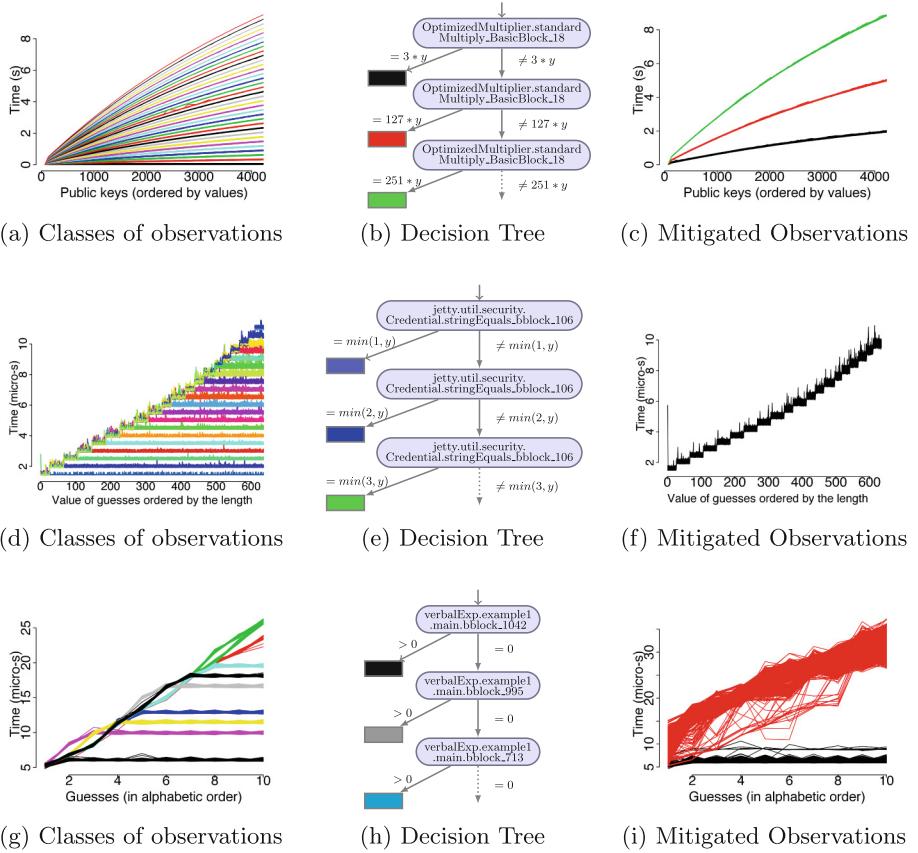


Fig. 6. Initial functional observations, decision tree, and the mitigated observations from left to right for Gabfeed, Jetty, and Verbal Expressions from top to bottom.

Fig. 6b. The monitoring system uses the decision tree model and automatically detects the current class of observation. Then, it adds extra delays based on the mitigation policy to enforce it. The results of the mitigation is shown in Fig. 6c. Answer for our research question. *Scalability:* It takes about 1 second to calculate the stochastic and the deterministic policies. *Security:* Stochastic and deterministic variants improve the min-guess entropy more than 100 times under the given performance overhead of 50%, respectively.

Findings for Jetty. The stochastic algorithm and the deterministic algorithm find the same policy that results in 1 cluster with 39.6% performance overhead. The min-guess entropy improves from 4.5 to 400.5. For the enforcement, SCHMIT first uses the initial clusterings and specifies their characteristics with program internals that result in the decision tree model shown in Fig. 6e. Since the response time is in the order of micro-seconds, we transform the source code using the decision tree model by adding extra counter variables. The results of

the mitigation is shown in Fig. 6f. *Scalability*: It takes less than 1 second to calculate the policies for both algorithms. *Security*: Stochastic and deterministic variants improve the min-guess entropy 89 times under the given performance overhead.

Findings for Java Verbal Expressions. For the stochastic algorithm, the policy results in 2 clusters, and the min-guess entropy has improved to 500.5. The performance overhead is 36%. For the dynamic programming, the policy results in 2 clusters. This adds 28% of performance overhead, while it improves the min-guess entropy from 50.5 to 450.5. The user chooses to use the deterministic policy for the mitigation. For the mitigation, we transform the source code using the decision tree model and add the extra delays based on the mitigation policy.

Findings for Password Matching. Both the deterministic and the stochastic algorithms result in finding a policy with 2 clusters where the min-guess entropy has improved from 1 to 5.5 with the performance overhead of 19.6%. For the mitigation, we transform the source code using the decision tree model and add extra delays based on the mitigation policy if necessary.

8 Related Work

Quantitative theory of information have been widely used to measure how much information is being leaked with side-channel observations [11, 20, 25, 41]. Mitigation techniques increase the remaining entropy of secret sets leaked through the side channels, while considering the performance [10, 23, 26, 40, 48, 49].

Köpf and Dürmuth [26] use a bucketing algorithm to partition programs' observations into intervals. With the unknown-message threat model, Köpf and Dürmuth [26] propose a dynamic programming algorithm to find the optimal number of possible observations under a performance penalty. The works [10, 48] introduce different black-box schemes to mitigate leaks. In particular, Askarov et al. [10] show the quantizing time techniques, which permit events to release at scheduled constant slots, have the worst case leakage if the slot is not filled with events. Instead, they introduce the double scheme method that has a schedule of predictions like the quantizing approach, but if the event source fails to deliver events at the predicted time, the failure results in generating a new schedule in which the interval between predictions is doubled. We compare our mitigation technique with both algorithms throughout this paper.

Elimination of timing side channels is a common technique to guarantee the confidentiality of software [7, 17, 27, 30, 31, 46]. The work [46] aims to eliminate side channels using static analysis enhanced with various techniques to keep the performance overheads low without guaranteeing the amounts of overhead. In contrast, we use dynamic analysis and allow a small amount of information to leak, but we guarantee an upper-bound on the performance overhead.

Machine learning techniques have been used for explaining timing differences between traces [42–44]. Tizpaz-Niari et al. [44] consider performance issues in softwares. They also cluster execution times of programs and then explain what

program properties distinguish the different functional clusters. We adopt their techniques for our security problem.

Acknowledgements. The authors would like to thank Mayur Naik for shepherding our paper and providing useful suggestions. This research was supported by DARPA under agreement FA8750-15-2-0096.

References

1. Branch and bound algorithm for mip problems. <http://www.gurobi.com/resources/getting-started/mip-basics>
2. Verbal expressions library. <https://github.com/VerbalExpressions/JavaVerbalExpressions>
3. Timing attack in google keyczar library (2009). <https://rdist.root.org/2009/05/28/timing-attack-in-google-keyczar-library/>
4. Gabfeed application (2016). https://github.com/Apogee-Research/STAC/tree/master/Engagement_Challenges/Engagement_2/gabfeed_1
5. Timing side-channel on the length of password in eclipse jetty May 2017. <https://github.com/eclipse/jetty.project/commit/2baa1abe4b1c380a30deacc1ed-367466a1a62ea>
6. Timing side-channel on the password in eclipse jetty May 2017. <https://github.com/eclipse/jetty.project/commit/f3751d70787fd8ab93932a51c60514c2eb37cb58>
7. Agat, J.: Transforming out timing leaks. In: Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 40–53. ACM (2000)
8. Aloise, D., Deshpande, A., Hansen, P., Popat, P.: Np-hardness of euclidean sum-of-squares clustering. *Mach. Learn.* **75**(2), 245–248 (2009)
9. Antonopoulos, T., Gazzillo, P., Hicks, M., Koskinen, E., Terauchi, T., Wei, S.: Decomposition instead of self-composition for proving the absence of timing channels. In: PLDI, pp. 362–375. ACM (2017)
10. Askarov, A., Zhang, D., Myers, A.C.: Predictive black-box mitigation of timing channels. In: Proceedings of the 17th ACM Conference on Computer and Communications Security, pp. 297–307. ACM (2010)
11. Backes, M., Köpf, B., Rybalchenko, A.: Automatic discovery and quantification of information leaks. In: 2009 30th IEEE Symposium on Security and Privacy, pp. 141–153. IEEE (2009)
12. Bertsekas, D.P.: Nonlinear programming. Athena Scientific, 2016. Tech. rep., ISBN 978-1-886529-05-2
13. Brumley, D., Boneh, D.: Remote timing attacks are practical. *Comput. Netw.* **48**(5), 701–716 (2005)
14. Chen, J., Feng, Y., Dillig, I.: Precise detection of side-channel vulnerabilities using quantitative cartesian hoare logic. In: CCS, pp. 875–890 (2017)
15. Chiba, S.: Javassist - a reflection-based programming wizard for java. In: Proceedings of OOPSLA 1998 Workshop on Reflective Programming in C++ and Java, vol. 174 (1998)
16. Dhem, J.-F., Koeune, F., Leroux, P.-A., Mestré, P., Quisquater, J.-J., Willems, J.-L.: A practical implementation of the timing attack. In: Quisquater, J.-J., Schneier, B. (eds.) CARDIS 1998. LNCS, vol. 1820, pp. 167–182. Springer, Heidelberg (2000). https://doi.org/10.1007/10721064_15

17. Eldib, H., Wang, C.: Synthesis of masking countermeasures against side channel attacks. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 114–130. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_8
18. Fallgren, M.: On the complexity of maximizing the minimum shannon capacity in wireless networks by joint channel assignment and power allocation. In: 2010 IEEE 18th International Workshop on Quality of Service (IWQoS), pp. 1–7 (2010)
19. Gurobi, L.: Optimization: Gurobi optimizer reference manual (2018). <http://www.gurobi.com>
20. Heusser, J., Malacaria, P.: Quantifying information leaks in software. In: Proceedings of the 26th Annual Computer Security Applications Conference, pp. 261–269. ACM (2010)
21. Jacques, J., Preda, C.: Functional data clustering: a survey. *Adv. Data Anal. Clasif.* **8**(3), 231–255 (2014)
22. Jones, E., Oliphant, T., Peterson, P., et al.: SciPy: open source scientific tools for Python (2001). <http://www.scipy.org/>
23. Kadloor, S., Kiyavash, N., Venkitasubramaniam, P.: Mitigating timing based information leakage in shared schedulers. In: 2012 Proceedings IEEE Infocom, pp. 1044–1052. IEEE (2012)
24. Kocher, P.C.: Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In: Koblitz, N. (ed.) CRYPTO 1996. LNCS, vol. 1109, pp. 104–113. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-68697-5_9
25. Köpf, B., Basin, D.: An information-theoretic model for adaptive side-channel attacks. In: Proceedings of the 14th ACM Conference on Computer and Communications Security, pp. 286–296. CCS 2007, ACM, New York (2007)
26. Köpf, B., Dürmuth, M.: A provably secure and efficient countermeasure against timing attacks. In: 22nd IEEE Computer Security Foundations Symposium, 2009, CSF 2009, pp. 324–335. IEEE (2009)
27. Köpf, B., Mantel, H.: Transformational typing and unification for automatically correcting insecure programs. *Int. J. Inf. Secur.* **6**(2–3), 107–131 (2007)
28. Korf, R.E.: A complete anytime algorithm for number partitioning. *AI* **106**, 181–203 (1998)
29. Lampson, B.W.: A note on the confinement problem. *Commun. ACM* **16**(10), 613–615 (1973)
30. Mantel, H., Starostin, A.: Transforming out timing leaks, more or less. In: Pernul, G., Ryan, P.Y.A., Weippl, E. (eds.) ESORICS 2015. LNCS, vol. 9326, pp. 447–467. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24174-6_23
31. Molnar, D., Piotrowski, M., Schultz, D., Wagner, D.: The program counter security model: automatic detection and removal of control-flow side channel attacks. In: Won, D.H., Kim, S. (eds.) ICISC 2005. LNCS, vol. 3935, pp. 156–168. Springer, Heidelberg (2006). https://doi.org/10.1007/11734727_14
32. Nemhauser, G.L., Wolsey, L.A.: Integer programming and combinatorial optimization. In: Nemhauser, G.L., Savelsbergh, M.W.P., Sigismondi, G.S. (1992). Constraint Classification for Mixed Integer Programming Formulations. COAL Bulletin, vol. 20, pp. 8–12. Wiley, Chichester (1988)
33. Nocedal, J., Wright, S.J.: Numerical Optimization 2nd (2006)
34. Nocedal, J., Wright, S.J.: Sequential Quadratic Programming. Springer, New York (2006)
35. Padlipsky, M., Snow, D., Karger, P.: Limitations of End-to-End Encryption in Secure Computer Networks. Tech. rep., MITRE CORP BEDFORD MA (1978)
36. Papadimitriou, C.H., Steiglitz, K.: Combinatorial Optimization: Algorithms and Complexity. Courier Corporation, North Chelmsford (1998)

37. Phan, Q.S., Bang, L., Pasareanu, C.S., Malacaria, P., Bultan, T.: Synthesis of adaptive side-channel attacks. In: 2017 IEEE 30th Computer Security Foundations Symposium (CSF), pp. 328–342. IEEE (2017)
38. Ramsay, J., Hooker, G., Graves, S.: Functional Data Analysis with R and MATLAB. Springer Science & Business Media, Berlin (2009)
39. Rivest, R.L., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. Commun. ACM **21**(2), 120–126 (1978)
40. Schinzel, S.: An efficient mitigation method for timing side channels on the web. In: 2nd International Workshop on Constructive Side-Channel Analysis and Secure Design (COSADE) (2011)
41. Smith, G.: On the foundations of quantitative information flow. In: de Alfaro, L. (ed.) FoSSaCS 2009. LNCS, vol. 5504, pp. 288–302. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00596-1_21
42. Song, L., Lu, S.: Statistical debugging for real-world performance problems. In: Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, pp. 561–578. OOPSLA 2014 (2014). <https://doi.org/10.1145/2660193.2660234>
43. Tizpaz-Niari, S., Černý, P., Chang, B.-Y.E., Sankaranarayanan, S., Trivedi, A.: Discriminating traces with time. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10206, pp. 21–37. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54580-5_2
44. Tizpaz-Niari, S., Černý, P., Chang, B.E., Trivedi, A.: Differential performance debugging with discriminant regression trees. In: 32nd AAAI Conference on Artificial Intelligence (AAAI), pp. 2468–2475 (2018)
45. Tizpaz-Niari, S., Černý, P., Trivedi, A.: Data-driven debugging for functional side channels. arXiv preprint. [arXiv:1808.10502](https://arxiv.org/abs/1808.10502) (2018)
46. Wu, M., Guo, S., Schaumont, P., Wang, C.: Eliminating timing side-channel leaks using program repair. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 15–26. ACM (2018)
47. Yarom, Y., Genkin, D., Heninger, N.: Cachebleed: a timing attack on openssl constant-time rsa. J. Cryptographic Eng. **7**(2), 99–112 (2017)
48. Zhang, D., Askarov, A., Myers, A.C.: Predictive mitigation of timing channels in interactive systems. In: Proceedings of the 18th ACM Conference on Computer and Communications Security, pp. 563–574. ACM (2011)
49. Zhang, D., Askarov, A., Myers, A.C.: Language-based control and mitigation of timing channels. PLDI **47**(6), 99–110 (2012)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Property Directed Self Composition

Ron Shemer¹⁽⁾, Arie Gurfinkel², Sharon Shoham¹, and Yakir Vizel³

¹ Tel Aviv University, Tel Aviv, Israel

ronsheme@mail.tau.ac.il

² University of Waterloo, Waterloo, Canada

³ The Technion, Haifa, Israel

Abstract. We address the problem of verifying *k-safety properties*: properties that refer to k interacting executions of a program. A prominent way to verify *k-safety* properties is by *self composition*. In this approach, the problem of checking *k-safety* over the original program is reduced to checking an “ordinary” safety property over a program that executes k copies of the original program in some order. The way in which the copies are composed determines how complicated it is to verify the composed program. We view this composition as provided by a *semantic self composition function* that maps each state of the composed program to the copies that make a move. Since the “quality” of a self composition function is measured by the ability to verify the safety of the composed program, we formulate the problem of inferring a self composition function together with the inductive invariant needed to verify safety of the composed program, where both are restricted to a given language. We develop a *property-directed* inference algorithm that, given a set of predicates, infers composition-invariant pairs expressed by Boolean combinations of the given predicates, or determines that no such pair exists. We implemented our algorithm and demonstrate that it is able to find self compositions that are beyond reach of existing tools.

1 Introduction

Many relational properties, such as noninterference [12], determinism [21], service level agreements [9], and more, can be reduced to the problem of *k-safety*. Namely, reasoning about k different traces of a program simultaneously. A common approach to verifying *k-safety* properties is by means of *self composition*, where the program is composed with k copies of itself [4, 32]. A state of the composed program consists of the states of each copy, and a trace naturally corresponds to k traces of the original program. Therefore, *k-safety* properties of the original program become ordinary safety properties of the composition, hence reducing *k-safety* verification to ordinary safety. This enables reasoning about *k-safety* properties using any of the existing techniques for safety verification such as Hoare logic [20] or model checking [7].

While self composition is sound and complete for *k-safety*, its applicability is questionable for two main reasons: (i) considering several copies of the program greatly increases the state space; and (ii) the way in which the different copies are composed when reducing the problem to safety verification affects the complexity of the resulting self composed program, and as such affects the complexity of verifying it. Improving the applicability of self composition has been the topic of many

works [2, 14, 18, 26, 30, 33]. However, most efforts are focused on compositions that are pre-defined, or only depend on syntactic similarities.

In this paper, we take a different approach; we build upon the observation that by choosing the “right” composition, the verification can be greatly simplified by leveraging “simple” correlations between the executions. To that end, we propose an algorithm, called PDSC, for inferring a *property directed* self composition. Our approach uses a *dynamic* composition, where the composition of the different copies can change during verification, directed at simplifying the verification of the composed program.

Compositions considered in previous work differ in the order in which the copies of the program execute: either synchronously, asynchronously, or in some mix of the two [3, 14, 34]. To allow general compositions, we define a *composition function* that maps every state of the composed program to the set of copies that are scheduled in the next step. This determines the order of execution for the different copies, and thus induces the self composed program. Unlike most previous works where the composition is pre-defined based on syntactic rules only, our composition is *semantic* as it is defined over the state of the composed program.

To capture the difficulty of verifying the composed program, we consider verification by means of inferring an inductive invariant, parameterized by a language for expressing the inductive invariant. Intuitively, the more expressive the language needs to be, the more difficult the verification task is. We then define the problem of inferring a composition function *together* with an inductive invariant for verifying the safety of the composed program, where both are restricted to a given language. Note that for a fixed language \mathcal{L} , an inductive invariant may exist for some composition function but not for another¹. Thus, the restriction to \mathcal{L} defines a target for the inference algorithm, which is now directed at finding a composition that admits an inductive invariant in \mathcal{L} .

Example 1. To demonstrate our approach, consider the program in Fig. 1. The program inserts a new value into an array. We assume that the array A and its length len are “low”-security variables, while the inserted value h is “high”-security. The first loop finds the location in which h will be inserted. Note that the number of iterations depends on the value of h . Due to that, the second loop executes to ensure that the output i (which corresponds to the number of iterations) does not leak sensitive data. As an example, we emphasize that without the second loop, i could leak the location of h in A . To express the property that i does not leak sensitive data, we use the 2-safety property that in any two executions, if the inputs A and len are the same, so is the output i .

To verify the 2-safety property, consider two copies of the program. Let the language \mathcal{L} for verifying the self composition be defined by the predicates depicted in Fig. 1. The most natural self composition to consider is a lock-step composition, where the copies execute synchronously. However, for such a composition the composed program may reach a state where, for example, $i_1 = i_2 + 1$. This occurs when the first copy exists the first loop, while the second copy is still executing it. Since the language cannot express this correlation between the two copies, no inductive invariant suffices to verify that $i_1 = i_2$ when the program terminates.

¹ See the extended version [29] for an example that requires a non-linear inductive invariant with a composition that is based on the control structure but has a linear invariant with another.

```

int arrayInsert(int[] A, int len, int h) {
    int i=0;
    1: while (i < len && A[i] < h)
        i++;
    2: len = shift_array(A, i, 1);
        A[i] = h;
    3: while (i < len)
        i++;
    4: return i;
}

composition:
if (pc1 < 3 && (pc2 > 0 || !cond1)
    && (pc2 == 3 || (pc2 == 0 && cond2)))
    step(1);
else if (pc2 < 3 && (pc1 > 0 || !cond2)
    && (pc1 == 3 || (pc1 == 0 && cond1)))
    step(2);
else step(1,2);

cond1 := i1 < len1 && A1[i1] < h1
cond2 := i2 < len2 && A2[i2] < h2

predicates: i1 = i2, i1 < len1, i2 < len2,
A1[i1] < h1, A2[i2] < h2, len1 = len2,
len1 = len2 + 1, len2 = len1 + 1

```

Fig. 1. Constant-time insert to an array.

In contrast, when verifying the 2-safety property, PDSC directs its search towards a composition function for which an inductive invariant in \mathcal{L} does exist. As such, it infers the composition function depicted in Fig. 1, as well as an inductive invariant in \mathcal{L} . The invariant for this composition implies that $i_1 = i_2$ at every state.

As demonstrated by the example, PDSC focuses on logical languages based on predicate abstraction [17], where inductive invariants can be inferred by model checking. In order to infer a composition function that admits an inductive invariant in \mathcal{L} , PDSC starts from a default composition function, and modifies its definition based on the reasoning performed by the model checker during verification. As the composition function is part of the verified model (recall that it is defined over the program state), different compositions are part of the state space explored by the model checker. As a result, a key ingredient of PDSC is identifying “bad” compositions that prevent it from finding an inductive invariant in \mathcal{L} . It is important to note that a naive algorithm that tries all possible composition functions has a time complexity $O(2^{2^{|P|}})$, where P is the set of predicates considered. However, integrating the search for a composition function into the model checking algorithm allows us to reduce the time complexity of the algorithm to $2^{O(|P|)}$, where we show that the problem is in fact PSPACE-hard.²

We implemented PDSC using SEAHORN [19], Z3 [25] and SPACER [22] and evaluated it on examples that demonstrate the need for nontrivial semantic compositions. Our results clearly show that PDSC can solve complex examples by inferring the required composition, while other tools cannot verify these examples. We emphasize that for these particular examples, lock-step composition is not sufficient. We also evaluated PDSC on the examples from [26, 30] that are proven with the trivial lock-step composition. On these examples, PDSC is comparable to state of the art tools.

Related Work. This paper addresses the problem of verifying k-safety properties (also called hyperproperties [8]) by means of self composition. Other approaches tackle the problem without self-composition, and often focus on more specific properties, most noticeably the 2-safety noninterference property (e.g. [1, 33]). Below we focus on works that use self-composition.

² Proofs of the claims made in this paper can be found in the extended version [29].

Previous work such as [2–4, 14, 15, 32] considered self composition (also called product programs) where the composition function is constant and set a-priori, using syntax-based hints. While useful in general, such self compositions may sometimes result in programs that are too complex to verify. This is in contrast to our approach, where the composition function is evolving during verification, and is adapted to the capabilities of the model checker.

The work most closely related to ours is [30] which introduces Cartesian Hoare Logic (CHL) for verification of k -safety properties, and designs a verification framework for this logic. This work is further improved in [26]. These works search for a proof in CHL, and in doing so, implicitly modify the composition. Our work infers the composition explicitly and can use off-the-shelf model checking tools. More importantly, when loops are involved both [30] and [26] use lock-step composition and align loops syntactically. Our algorithm, in contrast, does not rely on syntactic similarities, and can handle loops that cannot be aligned trivially.

There have been several results in the context of harnessing Constraint Horn Clauses (CHC) solvers for verification of relational properties [11, 24]. Given several copies of a CHC system, a product CHC system that synchronizes the different copies is created by a syntactical analysis of the rules in the CHC system. These works restrict the synchronization points to CHC predicates (i.e., program locations), and consider only one synchronization (obtained via transformations of the system of CHCs). On the other hand, our algorithm iteratively searches for a good synchronization (composition), and considers synchronizations that depend on program state.

Equivalence Checking and Regression Verification. Equivalence checking is another closely related research field, where a composition of several programs is considered. As an example, equivalence checking is applied to verify the correctness of compiler optimizations [10, 18, 28, 34]. In [28] the composition is determined by a brute-force search for possible synchronization points. While this brute-force search resembles our approach for finding the correct composition, it is not guided by the verification process. The works in [10, 18] identify possible synchronization points syntactically, and try to match them during the construction of a simulation relation between programs.

Regression verification also requires the ability to show equivalence between different versions of a program [15, 16, 31]. The problem of synchronizing unbalanced loops appears in [31] in the form of unbalanced recursive function calls. To allow synchronization in such cases, the user can specify different unrolling parameters for the different copies. In contrast, our approach relies only on user supplied predicates that are needed to establish correctness, while synchronization is handled automatically.

2 Preliminaries

In this paper we reason about programs by means of the transition systems defining their semantics. A transition system is a tuple $T = (S, R, F)$, where S is a set of states, $R \subseteq S \times S$ is a transition relation that specifies the steps in an execution of the program, and $F \subseteq S$ is a set of *terminal states* $F \subseteq S$ such that every terminal state $s \in F$ has an outgoing transition to itself and no additional transitions (terminal states allow us to

reason about pre/post specifications of programs). An *execution* or *trace* $\pi = s_0, s_1, \dots$ is a (finite or infinite) sequence of states such that for every $i \geq 0$, $(s_i, s_{i+1}) \in R$. The execution is *terminating* if there exists $0 \leq i \leq |\pi|$ such that $s_i \in F$. In this case, the suffix of the execution is of the form s_i, s_i, \dots and we say that π ends at s_i .

As usual, we represent transition systems using logical formulas over a set of variables, corresponding to the program variables. We denote the set of variables by \mathcal{V} . The set of terminal states is represented by a formula over \mathcal{V} and the transition relation is represented by a formula over $\mathcal{V} \uplus \mathcal{V}'$, where \mathcal{V} represents the pre-state of a transition and $\mathcal{V}' = \{v' \mid v \in \mathcal{V}\}$ represents its post-state. In the sequel, we use sets of states and their symbolic representation via formulas interchangeably.

Safety and Inductive Invariants. We consider safety properties defined via pre/post conditions.³ A *safety property* is a pair $(\text{pre}, \text{post})$ where pre, post are formulas over \mathcal{V} , representing subsets of S , denoting the pre- and post-condition, respectively. T satisfies $(\text{pre}, \text{post})$, denoted $T \models (\text{pre}, \text{post})$, if every terminating execution π of T that starts in a state s_0 such that $s_0 \models \text{pre}$ ends in a state s such that $s \models \text{post}$. In other words, for every state s that is reachable in T from a state in pre we have that $s \models F \rightarrow \text{post}$.

A prominent way to verify safety properties is by finding an inductive invariant. An *inductive invariant* for a transition system T and a safety property $(\text{pre}, \text{post})$ is a formula Inv such that (1) $\text{pre} \Rightarrow \text{Inv}$ (initiation), (2) $\text{Inv} \wedge R \Rightarrow \text{Inv}'$ (consecution), and (3) $\text{Inv} \Rightarrow (F \rightarrow \text{post})$ (safety), where $\varphi \Rightarrow \psi$ denotes the validity of $\varphi \rightarrow \psi$, and φ' denotes $\varphi(\mathcal{V}')$, i.e., the formula obtained after substituting every $v \in \mathcal{V}$ by the corresponding $v' \in \mathcal{V}'$. If there exists such an inductive invariant, then $T \models (\text{pre}, \text{post})$.

k-safety. A *k-safety property* refers to k interacting executions of T . Similarly to an ordinary property, it is defined by $(\text{pre}, \text{post})$, except that pre and post are defined over $\mathcal{V}^1 \uplus \dots \uplus \mathcal{V}^k$ where $\mathcal{V}^i = \{v^i \mid v \in \mathcal{V}\}$ denotes the i th copy of the program variables. As such, pre and post represent sets of k -tuples of program states (*k-states* for short): for a k -tuple (s_1, \dots, s_k) of states and a formula φ over $\mathcal{V}^1 \uplus \dots \uplus \mathcal{V}^k$, we say that $(s_1, \dots, s_k) \models \varphi$ if φ is satisfied when for each i , the assignment of \mathcal{V}^i is determined by s_i . We say that T satisfies $(\text{pre}, \text{post})$, denoted $T \models^k (\text{pre}, \text{post})$, if for every k terminating executions π^1, \dots, π^k of T that start in states s_1, \dots, s_k , respectively, such that $(s_1, \dots, s_k) \models \text{pre}$, it holds that they end in states t_1, \dots, t_k , respectively, such that $(t_1, \dots, t_k) \models \text{post}$.

For example, the *non interference* property may be specified by the following 2-safety property: $\text{pre} = \bigwedge_{v \in \text{LowIn}} v^1 = v^2$, $\text{post} = \bigwedge_{v \in \text{LowOut}} v^1 = v^2$ where LowIn and LowOut denote subsets of the program inputs, resp. outputs, that are considered “low security” and the rest are classified as “high security”. This property asserts that every 2 terminating executions that start in states that agree on the “low security” inputs end in states that agree on the low security outputs, i.e., the outcome does not depend on any “high security” input and, hence, does not leak secure information.

Checking k -safety properties reduces to checking ordinary safety properties by creating a *self composed program* that consists of k copies of the transition system, each

³ Our results can be extended to arbitrary safety (and k -safety) properties by introducing “observable” states to which the property may refer.

with its own copy of the variables, that run in parallel in some way. Thus, the self composed program is defined over variables $\mathcal{V}^{\parallel k} = \mathcal{V}^1 \uplus \dots \uplus \mathcal{V}^k$, where $\mathcal{V}^i = \{v^i \mid v \in \mathcal{V}\}$ denotes the variables associated with the i th copy. For example, a common composition is a *lock-step* composition in which the copies execute simultaneously. The resulting composed transition system $T^{\parallel k} = (S^{\parallel k}, R^{\parallel k}, F^{\parallel k})$ is defined such that $S^{\parallel k} = S \times \dots \times S$, $F^{\parallel k} = \bigwedge_{i=1}^k F(\mathcal{V}^i)$ and $R^{\parallel k} = \bigwedge_{i=1}^k R(\mathcal{V}^j, \mathcal{V}^{j'})$. Note that $R^{\parallel k}$ is defined over $\mathcal{V}^{\parallel k} \uplus \mathcal{V}^{\parallel k'}$ (as usual). Then, the k -safety property (*pre, post*) is satisfied by T if and only if an ordinary safety property (*pre, post*) is satisfied by $T^{\parallel k}$. More general notions of *self composition* are investigated in Sect. 3.

3 Inferring Self Compositions for Restricted Languages of Inductive Invariants

Any self-composition is sufficient for reducing k -safety to safety, e.g., lock-step, sequential, synchronous, asynchronous, etc. However, the choice of the self-composition used determines the difficulty of the resulting safety problem. Different self composed programs would require different inductive invariants, some of which cannot be expressed in a given logical language.

In this section, we formulate the problem of inferring a self composition function such that the obtained self composed program may be verified with a given language of inductive invariants. We are, therefore, interested in inferring both the self composition function and the inductive invariant for verifying the resulting self composed program. We start by formulating the kind of self compositions that we consider.

In the sequel, we fix a transition system $T = (S, R, F)$ with a set of variables \mathcal{V} .

3.1 Semantic Self Composition

Roughly speaking, a k self composition of T consists of k copies of T that execute together in some order, where steps may interleave or be performed simultaneously. The order is determined by a self composition function, which may also be viewed as a scheduler that is responsible for scheduling a subset of the copies in each step. We consider *semantic* compositions in which the order may depend on the *states* of the different copies, as well as the correlations between them (as opposed to *syntactic* compositions that only depend on the control locations of the copies, but may not depend on the values of other variables):

Definition 1 (Semantic Self Composition Function). A semantic k self composition function (*k -composition function for short*) is a function $f : S^k \rightarrow \mathbb{P}(\{1..k\})$, mapping each k -state to a nonempty set of copies that are to participate in the next step of the self composed program⁴.

⁴ We consider *memoryless* composition functions. Compositions that depend on the history of the (joint) execution are supported via ghost state added to the program to track the history.

We represent a k -composition function f by a set of logical conditions, with a condition C_M for every nonempty subset $M \subseteq \{1..k\}$ of the copies. For each such $M \subseteq \{1..k\}$, the condition C_M is defined over $\mathcal{V}^{\parallel k} = \mathcal{V}^1 \uplus \dots \uplus \mathcal{V}^k$, and hence it represents a set of k -states, with the meaning that all the k -states that satisfy C_M are mapped to M by f :

$$f(s_1, \dots, s_k) = M \text{ if and only if } (s_1, \dots, s_k) \models C_M.$$

To ensure that the function is well defined, we require that $(\bigvee_M C_M) \equiv \text{true}$, which ensures that every k -state satisfies at least one of the conditions. We also require that for every $M_1 \neq M_2$, $C_{M_1} \wedge C_{M_2} \equiv \text{false}$, hence every k -state satisfies at most one condition. Together these requirements ensure that the conditions induce a partition of the set of all k -states. In the sequel, we identify a k -composition function f with its symbolic representation via conditions $\{C_M\}_M$ and use them interchangeably.

Definition 2 (Composed Program). *Given a k -composition function f , represented via conditions C_M for every nonempty set $M \subseteq \{1..k\}$, we define the k self composition of T to be the transition system $T^f = (S^{\parallel k}, R^f, F^{\parallel k})$ over variables $\mathcal{V}^{\parallel k} = \mathcal{V}^1 \uplus \dots \uplus \mathcal{V}^k$ defined as follows: $F^{\parallel k} = \bigwedge_{i=1}^k F^i$, where $F^i = F(\mathcal{V}^i)$, and*

$$R^f = \bigvee_{\emptyset \neq M \subseteq \{1..k\}} (C_M \wedge \varphi_M) \quad \text{where} \quad \varphi_M = \bigwedge_{j \in M} R(\mathcal{V}^j, \mathcal{V}^{j'}) \wedge \bigwedge_{j \notin M} \mathcal{V}^j = \mathcal{V}^{j'}$$

Thus, in T^f , the set of states consists of k -states ($S^{\parallel k} = S \times \dots \times S$), the terminal states are k -states in which all the individual states are terminal, and the transition relation includes a transition from (s_1, \dots, s_k) to (s'_1, \dots, s'_k) if and only if $f(s_1, \dots, s_k) = M$ and $(\forall i \in M. (s_i, s'_i) \in R) \wedge (\forall i \notin M. s_i = s'_i)$. That is, every transition of T^f corresponds to a simultaneous transition of a subset M of the k copies of T , where the subset is determined by the self composition function f . If $f(s_1, \dots, s_k) = M$, then for every $i \in M$ we say that i is *scheduled* in (s_1, \dots, s_k) .

Example 2. A k self composition that runs the k copies of T sequentially, one after the other, corresponds to a k -composition function f defined by $f(s_1, \dots, s_k) = \{i\}$ where $i \in \{1..k\}$ is the minimal index of a non-terminal state in $\{s_1, \dots, s_k\}$. If all states in $\{s_1, \dots, s_k\}$ are terminal then $i = k$ (or any other index). This is encoded as follows: for every $1 \leq i < k$, $C_{\{i\}} = \neg F^i \wedge \bigwedge_{j < i} F^j$, $C_{\{k\}} = \bigwedge_{j < k} F^j$ and $C_M = \text{false}$ for every other $M \subseteq \{1..k\}$.

Example 3. The lock-step composition that runs the k copies of T synchronously corresponds to a k -self composition function f defined by $f(s_1, \dots, s_k) = \{1, \dots, k\}$, and encoded by $C_{\{1, \dots, k\}} = \text{true}$ and $C_M = \text{false}$ for every other $M \subseteq \{1..k\}$.

In order to ensure soundness of a reduction of k -safety to safety via self composition, one has to require that the self composition function does not “starve” any copy of the transition system that is about to terminate if it continues to execute. We refer to this requirement as *fairness*.

Definition 3 (Fairness). A k -self composition function f is fair iff for every k terminating executions π^1, \dots, π^k of T there exists an execution π^{\parallel} of T^f such that for every copy $i \in \{1..k\}$, the projection of π^{\parallel} to i is π^i .

Note that by the definition of the terminal states of T^f , π^{\parallel} as above is guaranteed to be terminating. We say that the i th copy *terminates* in π^{\parallel} if π^{\parallel} contains a k -state (s_1, \dots, s_k) such that $s_i \in F$. Fairness may be enforced in a straightforward way by requiring that whenever $f(s_1, \dots, s_k) = M$, the set M includes no index i for which $s_i \in F$, unless all have terminated. Since we assume that terminal states may only transition to themselves, a weaker requirement that suffices to ensure fairness is that M includes at least one index i for which $s_i \notin F$, unless there is no such index.

The following claim is now straightforward:

Lemma 1. Let T be a transition system, $(\text{pre}, \text{post})$ a k -safety property, and f a fair k -composition function for T and $(\text{pre}, \text{post})$. Then

$$T \models^k (\text{pre}, \text{post}) \text{ iff } T^f \models (\text{pre}, \text{post}).$$

Proof (sketch). Every terminating execution of T^f corresponds to k terminating executions of T . Fairness of f ensures that the converse also holds.

To demonstrate the necessity of the fairness requirement, consider a (non-fair) self composition function f that maps every state to $\{1\}$. Then, regardless of what the actual transition system T does, the resulting self composition T^f satisfies every pre-post specification vacuously, as it never reaches a terminal state.

Remark 1. While we require the conditions $\{C_M\}_M$ defining a self composition function f to induce a partition of $S^{\parallel k}$ in order to ensure that f is well defined as a (total) function, the requirement may be relaxed in two ways. First, we may allow C_{M_1} and C_{M_2} to overlap. This will add more transitions and may make the task of verifying the composed program more difficult, but it maintains the soundness of the reduction. Second, it suffices that the conditions cover the set of *reachable states* of the composed program rather than the entire state space. These relaxations do not damage soundness. Technically, this means that f represented by the conditions is a relation rather than a function. We still refer to it as a function and write $f(s_1, \dots, s_k) = M$ to indicate that $(s_1, \dots, s_k) \models C_M$, not excluding the possibility that $(s_1, \dots, s_k) \models M'$ for $M' \neq M$ as well. We note that as long as the language used to describe compositions is closed under Boolean operations, we can always extract from the conditions $\{C_M\}_M$ a function f' . This is done as follows: First, to prevent the overlap between conditions, determine an arbitrary total order $<$ on the sets $M \subseteq \{1..k\}$ and set $C'_M := C_M \wedge \bigwedge_{N < M} \neg C_N$. Second, to ensure that the conditions cover the entire state space, set $C'_{\{1..k\}} := C'_{\{1..k\}} \vee \neg(\bigvee_M C_M)$. It is easy to verify that f' defined by $\{C'_M\}_M$ is a total self composition function and that if f is fair, then so is f' .

3.2 The Problem of Inferring Self Composition with Inductive Invariant

Lemma 1 states the soundness of the reduction of k -safety to ordinary safety. Together with the ability to verify safety by means of an inductive invariant, this leads to a verification procedure. However, while soundness of the reduction holds for *any* self composition, an inductive invariant in a given language may exist for the composed program

resulting from some compositions but not from others. We therefore consider the self composition function and the inductive invariant together, as a pair, leading to the following definition.

Definition 4. Let T be a transition system and $(\text{pre}, \text{post})$ a k safety property. For a formula Inv over $\mathcal{V}^{\parallel k}$ and a self composition function f represented by conditions $\{C_M\}_M$, we say that (f, Inv) is a composition-invariant pair for T and $(\text{pre}, \text{post})$ if the following conditions hold:

- $\text{pre} \implies \text{Inv}$ (initiation of Inv),
- for every $\emptyset \neq M \subseteq \{1..k\}$, $\text{Inv} \wedge C_M \wedge \varphi_M \implies \text{Inv}'$ (consecution of Inv for R^f),
- $\text{Inv} \implies ((\bigwedge_{j=1}^k F^j) \rightarrow \text{post})$ (safety of Inv),
- $\text{Inv} \implies \bigvee_M C_M$ (f covers the reachable states),
- for every $\emptyset \neq M \subseteq \{1..k\}$, $C_M \wedge (\bigvee_{j=1}^k \neg F^j) \implies \bigvee_{j \in M} \neg F^j$ (f is fair).

As commented in Remark 1, we relax the requirement that $(\bigvee_M C_M) \equiv \text{true}$ to $\text{Inv} \implies \bigvee_M C_M$, thus ensuring that the conditions cover all the reachable states. Since the reachable states of T^f are determined by $\{C_M\}_M$ (which define f), this reveals the interplay between the self composition function and the inductive invariant. Furthermore, we do not require that $C_{M_1} \wedge C_{M_2} \equiv \text{false}$ for $M_1 \neq M_2$, hence a k -state may satisfy multiple conditions. As explained earlier, these relaxations do not damage soundness. Furthermore, if we construct from f a self composition function f' as described in Remark 1, Inv would be an inductive invariant for $T^{f'}$ as well.

Lemma 2. If there exists a composition-invariant pair (f, Inv) for T and $(\text{pre}, \text{post})$, then $T \models^k (\text{pre}, \text{post})$.

If we do not restrict the language in which f and Inv are specified, then the converse also holds. However, in the sequel we are interested in the ability to verify k -safety with a given language, e.g., one for which the conditions of Definition 4 belong to a decidable fragment of logic and hence can be discharged automatically.

Definition 5 (Inference in \mathcal{L}). Let \mathcal{L} be a logical language. The problem of inferring a composition-invariant pair in \mathcal{L} is defined as follows. The input is a transition system T and a k -safety property $(\text{pre}, \text{post})$. The output is a composition-invariant pair (f, Inv) for T and $(\text{pre}, \text{post})$ (as defined in Definition 4), where $\text{Inv} \in \mathcal{L}$ and f is represented by conditions $\{C_M\}_M$ such that $C_M \in \mathcal{L}$ for every $\emptyset \neq M \subseteq \{1..k\}$. If no such pair exists, the output is “no solution”.

When no solution exists, it does not necessarily mean that $T \not\models^k (\text{pre}, \text{post})$. Instead, it may be that the language \mathcal{L} is simply not expressive enough. Unfortunately, for expressive languages (e.g., quantified formulas or even quantifier free linear integer arithmetic), the problem of inferring an inductive invariant alone is already undecidable, making the problem of inferring a composition-invariant pair undecidable as well:

Lemma 3. Let \mathcal{L} be closed under Boolean operations and under substitution of a variable with a value, and include equalities of the form $v = a$, where v is a variable and a is a value (of the same sort). If the problem of inferring an inductive invariant in \mathcal{L} is undecidable, then so is the problem of inferring a composition-invariant pair in \mathcal{L} .

For example, linear integer arithmetic satisfies the conditions of the lemma. This motivates us to restrict the languages of inductive invariants. Specifically, we consider languages defined by a finite set of predicates. We consider *relational* predicates, defined over $\mathcal{V}^{\parallel k} = \mathcal{V}^1 \uplus \dots \uplus \mathcal{V}^k$. For a finite set of predicates \mathcal{P} , we define $\mathcal{L}_{\mathcal{P}}$ to be the set of all formulas obtained by Boolean combinations of the predicates in \mathcal{P} .

Definition 6 (Inference using predicate abstraction). *The problem of inferring a predicate-based composition-invariant pair is defined as follows. The input is a transition system T , a k -safety property $(pre, post)$, and a finite set of predicates \mathcal{P} . The output is the solution to the problem of inferring a composition-invariant pair for T and $(pre, post)$ in $\mathcal{L}_{\mathcal{P}}$.*

Remark 2. It is possible to decouple the language used for expressing the self composition function from the language used to express the inductive invariant. Clearly, different sets of predicates (and hence languages) can be assigned to the self composition function and to the inductive invariant. However, since inductiveness is defined with respect to the transitions of the composed system, which are in turn defined by the self composition function, if the language defining f is not included in the language defining Inv , the conditions C_M themselves would be over-approximated when checking the requirements of Definition 4 and therefore would incur a precision loss. For this reason, we use the same language for both.

Since the problem of invariant inference in $\mathcal{L}_{\mathcal{P}}$ is PSPACE-hard [23], a reduction from the problem of inferring inductive invariants to the problem of inferring composition-invariant pairs (similar to the one used in the proof of Lemma 3) shows that composition-invariant inference in $\mathcal{L}_{\mathcal{P}}$ is also PSPACE-hard:

Theorem 1. *Inferring a predicate-based composition-invariant pair is PSPACE-hard.*

4 Algorithm for Inferring Composition-Invariant Pairs

In this section, we present Property Directed Self-Composition, PDSC for short—our algorithm for tackling the composition-invariant inference problem for languages of predicates (Definition 6). Namely, given a transition system T , a k -safety property $(pre, post)$ and a finite set of predicates \mathcal{P} , we address the problem of finding a pair (f, Inv) , where f is a self composition function and Inv is an inductive invariant for the composed transition system T^f obtained from f , and both of them are in $\mathcal{L}_{\mathcal{P}}$, i.e., defined by Boolean combinations of the predicates in \mathcal{P} .

We rely on the property that a transition system (in our case T^f) has an inductive invariant in $\mathcal{L}_{\mathcal{P}}$ if and only if its abstraction obtained using \mathcal{P} is safe. This is because, the set of reachable abstract states is the strongest set expressible in $\mathcal{L}_{\mathcal{P}}$ that satisfies initiation and consecution. Given T^f , this allows us to use predicate abstraction to either obtain an inductive invariant in $\mathcal{L}_{\mathcal{P}}$ for T^f (if the abstraction of T^f is safe) or determine that no such inductive invariant exists (if an abstract counterexample trace is obtained). The latter indicates that a different self composition function needs to be considered. A naive realization of this idea gives rise to an iterative algorithm that starts from an

```

1  $f \leftarrow \text{lockstep}, E \leftarrow \emptyset, \text{Unreach} \leftarrow \text{false}$ 
2 while (true) do
3    $(res, Inv, cex) \leftarrow \text{Abs\_Reach}(\mathcal{P}, T^f, pre, post, \text{Unreach})$ 
4   if res = safe then return (f, Inv( $\mathcal{P}$ ))
5    $(\hat{s}, M) \leftarrow \text{Last\_Step}(cex)$ 
6    $E \leftarrow E \cup \{(\hat{s}, M)\}$ 
7   while (All_Excluded_Or_Starving( $\hat{s}, E$ )) do
8      $\text{Unreach} \leftarrow \text{Unreach} \vee \hat{s}$ 
9     if Unreach  $\wedge \varphi_{pre}(\mathcal{B}) \not\models \text{false}$  then return “no solution in  $\mathcal{L}_{\mathcal{P}}$ ”
10     $cex \leftarrow \text{Remove\_Last\_Step}(cex)$ 
11     $(\hat{s}, M) \leftarrow \text{Last\_Step}(cex)$ 
12     $E \leftarrow E \cup \{(\hat{s}, M)\}$ 
13   $f \leftarrow \text{Modify\_SC}(f, \hat{s}, E)$ 

```

Algorithm 1. PDSC: Property-Directed Self-Composition.

arbitrary initial composition function and in each iteration computes a new composition function. At the worst case such an algorithm enumerates all self composition functions defined in $\mathcal{L}_{\mathcal{P}}$, i.e., has time complexity $O(2^{2^{|\mathcal{P}|}})$. Importantly, we observe that, when no inductive invariant exists for some composition function, we can use the abstract counterexample trace returned in this case to (i) generalize and eliminate multiple composition functions, and (ii) identify that some abstract states must be unreachable if there is to be a composition-invariant pair, i.e., we “block” states in the spirit of *property directed reachability* [5, 13]. This leads to the algorithm depicted in Algorithm 1 whose worst case time complexity is $2^{O(|\mathcal{P}|)}$. Next, we explain the algorithm in detail.

Finding an Inductive Invariant for a Given Composition Function Using Predicate Abstraction. We use predicate abstraction [17, 27] to check if a given candidate composition function has a corresponding inductive invariant. This is done as follows. The abstraction of T^f using \mathcal{P} , denoted $A_{\mathcal{P}}(T^f)$, is a transition system (\hat{S}, \hat{R}) defined over variables \mathcal{B} , where $\mathcal{B} = \{b_p \mid p \in \mathcal{P}\}$ (we omit the terminal states). $\hat{S} = \{0, 1\}^{\mathcal{B}}$, i.e., each abstract state corresponds to a valuation of the Boolean variables representing \mathcal{P} . An abstract state $\hat{s} \in \hat{S}$ represents the following set of states of T^f :

$$\gamma(\hat{s}) = \{s^{\parallel} \in S^{\parallel k} \mid \forall p \in \mathcal{P}. s^{\parallel} \models p \Leftrightarrow \hat{s}(b_p) = 1\}$$

We extend γ to sets of states and to formulas representing sets of states in the usual way. The abstract transition relation is defined as usual:

$$\hat{R} = \{(\hat{s}_1, \hat{s}_2) \mid \exists s^{\parallel}_1 \in \gamma(\hat{s}_1) \exists s^{\parallel}_2 \in \gamma(\hat{s}_2). (s^{\parallel}_1, s^{\parallel}_2) \in R^f\}$$

Note that the set of abstract states in $A_{\mathcal{P}}(T^f)$ does *not* depend on f .

Notation. We sometimes refer to an abstract state $\hat{s} \in \hat{S}$ as the formula $\bigwedge_{\hat{s}(b_p)=1} b_p \wedge \bigwedge_{\hat{s}(b_p)=0} \neg b_p$. For a formula $\psi \in \mathcal{L}_{\mathcal{P}}$, we denote by $\psi(\mathcal{B})$ the result of substituting each $p \in \mathcal{P}$ in ψ by the corresponding Boolean variable b_p . For the opposite direction, given

a formula ψ over \mathcal{B} , we denote by $\psi(\mathcal{P})$ the formula in $\mathcal{L}_{\mathcal{P}}$ resulting from substituting each $b_p \in \mathcal{B}$ in ψ by p . Therefore, $\psi(\mathcal{P})$ is a symbolic representation of $\gamma(\psi)$.

Every set defined by a formula $\psi \in \mathcal{L}_{\mathcal{P}}$ is precisely represented by $\psi(\mathcal{B})$ in the sense that $\gamma(\psi(\mathcal{B}))$ is equal to the set of states defined by ψ , i.e., $\psi(\mathcal{B})$ is a precise abstraction of ψ . For simplicity, we assume that the termination conditions as well as the pre/post specification can be expressed precisely using the abstraction, in the following sense:

Definition 7. \mathcal{P} is adequate for T and (pre, post) if there exist $\varphi_{pre}, \varphi_{post}, \varphi_{F^i} \in \mathcal{L}_{\mathcal{P}}$ such that $\varphi_{pre} \equiv \text{pre}$, $\varphi_{post} \equiv \text{post}$ and $\varphi_{F^i} \equiv F^i$ (for every copy $i \in \{1..k\}$).

The following lemma provides the foundation for our algorithm:

Lemma 4. Let T be a transition system, (pre, post) a k safety property, and \mathcal{P} a finite set of predicates adequate for T and (pre, post). For a self composition function f defined via conditions $\{C_M\}_M$ in $\mathcal{L}_{\mathcal{P}}$, there exists an inductive invariant Inv in $\mathcal{L}_{\mathcal{P}}$ such that (f, Inv) is a composition-invariant pair for T and (pre, post) if and only if the following three conditions hold:

- S1** All reachable states of $A_{\mathcal{P}}(T^f)$ from $\varphi_{pre}(\mathcal{B})$ satisfy $(\bigwedge_{i=1}^k \varphi_{F^i}(\mathcal{B})) \rightarrow \varphi_{post}(\mathcal{B})$,
- S2** All reachable states of $A_{\mathcal{P}}(T^f)$ from $\varphi_{pre}(\mathcal{B})$ satisfy $\bigvee_M C_M(\mathcal{B})$, and
- S3** For every $\emptyset \neq M \subseteq \{1..k\}$, $C_M(\mathcal{B}) \wedge (\bigvee_{j=1}^k \neg \varphi_{F^j}(\mathcal{B})) \implies \bigvee_{j \in M} \neg \varphi_{F^j}(\mathcal{B})$.

Furthermore, if the conditions hold, then the symbolic representation of the set of abstract states of $A_{\mathcal{P}}(T^f)$ reachable from $\varphi_{pre}(\mathcal{B})$ is a formula Inv over \mathcal{B} such that $(f, Inv(\mathcal{P}))$ is a composition-invariant pair for T and (pre, post).

Algorithm 1 starts from the lock-step self composition function (Line 1), which is fair⁵, and constructs the next candidate f such that condition S3 in Lemma 4 always holds (see discussion of `Modify_SC`). Thus, condition S3 need not be checked explicitly.

Algorithm 1 checks whether conditions S1 and S2 hold for a given candidate composition function f by calling `Abs_Reach` (Line.3) – both checks are performed via a (non-)reachability check in $A_{\mathcal{P}}(T^f)$, checking whether a state violating $(\bigwedge_{i=1}^k \varphi_{F^i}(\mathcal{B})) \rightarrow \varphi_{post}(\mathcal{B})$ or $\bigvee_M C_M(\mathcal{B})$ is reachable from $\varphi_{pre}(\mathcal{B})$. Algorithm 1 maintains the abstract states that are not in $\bigvee_M C_M(\mathcal{B})$ by the formula *Unreach* defined over \mathcal{B} , which is initialized to *false* (as the lock-step composition function is defined for every state) and is updated in each iteration of Algorithm 1 to include the abstract states violating $\bigvee_M C_M(\mathcal{B})$. If no abstract state violating S1 or S2 is reachable, i.e., the conditions hold, then `Abs_Reach` returns the (potentially overapproximated) set of reachable abstract states, represented by a formula *Inv* over \mathcal{B} . In this case, by Lemma 4, $(f, Inv(\mathcal{P}))$ is a composition-invariant pair (line 4). Otherwise, an abstract counterexample is obtained. (We can of course apply bounded model checking to check if the counterexample is real; we omit this check as our focus is on the case where the system is safe.)

Remark 3. In practice, we do not construct $A_{\mathcal{P}}(T^f)$ explicitly. Instead, we use the implicit predicate abstraction approach [6].

⁵ Any fair self composition can be chosen as the initial one; we chose lock-step since it is a good starting point in many applications.

Eliminating Self Composition Candidates Based on Abstract Counterexamples. An abstract counterexample to conditions **S1** or **S2** indicates that the candidate composition function f has no corresponding *Inv*. Violation of **S1** can only be resolved by changing f such that the abstract trace is no longer feasible. Violation of **S2** may, in principle, also be resolved by extending the definition of f such that it is defined for all the abstract states in the counterexample trace.

However, to prevent the need to explore both options, our algorithm maintains the following invariant for every candidate self composition function f that it constructs:

Claim. Every abstract state that is *not* in $\bigvee_M C_M(\mathcal{B})$ is not reachable w.r.t. the abstract composed program of *any* composition function that is part of a composition-invariant pair for T and $(\text{pre}, \text{post})$.

This property clearly holds for the lock-step composition function, which the algorithm starts with, since for this composition, $\bigvee_M C_M(\mathcal{B}) \equiv \text{true}$. As we explain in Corollary 2, it continues to hold throughout the algorithm.

As a result of this property, whenever a candidate composition function f does not satisfy condition **S1** or **S2**, it is never the case that $\bigvee_M C_M(\mathcal{B})$ needs to be extended to allow the abstract states in *cex* to be reachable. Instead, the abstract counterexample obtained in violation of the conditions needs to be eliminated by modifying f .

Let $cex = \hat{s}_1, \dots, \hat{s}_{m+1}$ be an abstract counterexample of $A_{\mathcal{P}}(T^f)$ such that $\hat{s}_1 \models \varphi_{\text{pre}}(\mathcal{B})$ and $\hat{s}_{m+1} \models (\bigwedge_{i=1}^k \varphi_{F^i}(\mathcal{B})) \wedge \neg \varphi_{\text{post}}(\mathcal{B})$ (violating **S1**) or $\hat{s}_{m+1} \models \text{Unreach}$ (violating **S2**). Any self composition f' that agrees with f on the states in $\gamma(\hat{s}_i)$ for every \hat{s}_i that appears in *cex* has the same transitions in R^f and, hence, the same transitions in \hat{R} . It, therefore, exhibits the same abstract counterexample in $A_{\mathcal{P}}(T^{f'})$. Hence, it violates **S1** or **S2** and is not part of any composition-invariant pair.

Notation. Recall that f is defined via conditions $C_M \in \mathcal{L}_{\mathcal{P}}$. This ensures that for every abstract state \hat{s} , f is defined in the same way for all the states in $\gamma(\hat{s})$. We denote the value of f on the states in $\gamma(\hat{s})$ by $f(\hat{s})$ (in particular, $f(\hat{s})$ may be undefined). We get that $f(\hat{s}) = M$ if and only if $\hat{s} \models C_M(\mathcal{B})$.

Using this notation, to eliminate the abstract counterexample *cex*, one needs to eliminate at least one of the transitions in *cex* by changing the definition of $f(\hat{s}_i)$ for *some* $1 \leq i \leq m$. For a new candidate function f' this may be encoded by the disjunctive constraint $\bigvee_{i=1}^m f'(\hat{s}_i) \neq f(\hat{s}_i)$. However, we observe that a stronger requirement may be derived from *cex* based on the following lemma:

Lemma 5. Let f be a self composition function and $cex = \hat{s}_1, \dots, \hat{s}_{m+1}$ a counterexample trace in $A_{\mathcal{P}}(T^f)$ such that $\hat{s}_1 \models \varphi_{\text{pre}}(\mathcal{B})$ but $\hat{s}_{m+1} \models (\bigwedge_{i=1}^k \varphi_{F^i}(\mathcal{B})) \wedge \neg \varphi_{\text{post}}(\mathcal{B})$ or $\hat{s}_{m+1} \models \text{Unreach}$. Then for any self composition function f' such that $f'(\hat{s}_m) = f(\hat{s}_m)$, if \hat{s}_m is reachable in $A_{\mathcal{P}}(T^{f'})$ from $\varphi_{\text{pre}}(\mathcal{B})$, then a counterexample trace to **S1** or **S2** exists.

Corollary 1. If there exists a composition-invariant pair (f', Inv') , then there is also one where $f'(\hat{s}_m) \neq f(\hat{s}_m)$.

Therefore, we require that in the next self composition candidates the abstract state \hat{s}_m must not be mapped to its current value in f , i.e., $f'(\hat{s}_m) \neq M$, where $f(\hat{s}_m) = M$ ⁶.

Algorithm 1 accumulates these constraints in the set E (Line 6). Formally, the constraint $(\hat{s}, M) \in E$ asserts that C'_M must imply $\neg(\bigwedge_{\hat{s}(b_p)=1} p \wedge \bigwedge_{\hat{s}(b_p)=0} \neg p)$, and hence $f'(\hat{s}) \neq M$.

Identifying Abstract States that Must Be Unreachable. A new candidate self composition is constructed such that it satisfies all the constraints in E (thus ensuring that no abstract counterexample will re-appear). In the construction, we make sure to satisfy **S3** (fairness). Therefore, for every abstract state \hat{s} , we choose a value $f'(\hat{s})$ that satisfies the constraints in E and is *non-starving*: a value M is starving for \hat{s} if $\hat{s} \models \bigvee_{j=1}^k \neg \varphi_{F^j}(\mathcal{B})$ but $\hat{s} \not\models \bigvee_{j \in M} \neg \varphi_{F^j}(\mathcal{B})$, i.e., some of the copies have not terminated in \hat{s} but none of the non-terminating copies is scheduled. (Due to adequacy, a value M is starving for \hat{s} if and only if it is starving for every $s^\parallel \in \gamma(\hat{s})$.)

If for some abstract state \hat{s} , all the non-starving values have already been excluded (i.e., $(\hat{s}, M) \in E$ for every non-starving M), we conclude that there is *no* f' such that \hat{s} is reachable in $A_{\mathcal{P}}(T^{f'})$ and f' is part of a composition-invariant pair:

Lemma 6. *Let $\hat{s} \in \hat{S}$ be an abstract state such that for every $\emptyset \neq M \subseteq \{1..k\}$ either M is starving for \hat{s} or $(\hat{s}, M) \in E$. Then, for every f' that satisfies **S3**, if $A_{\mathcal{P}}(T^{f'})$ satisfies **S1** and **S2**, then \hat{s} is unreachable in $A_{\mathcal{P}}(T^{f'})$.*

Corollary 2. *If there exists a composition-invariant pair (f', Inv') , then \hat{s} is unreachable in $A_{\mathcal{P}}(T^{f'})$.*

This is because no matter how the self composition function f' would be defined, \hat{s} is guaranteed to have an outgoing abstract counterexample trace in $A_{\mathcal{P}}(T^{f'})$.

We, therefore, turn $f'(\hat{s})$ to be undefined. As a result, condition **S2** of Algorithm 4 requires that \hat{s} will be unreachable in $A_{\mathcal{P}}(T^{f'})$. In Algorithm 1, this is enforced by adding \hat{s} to *Unreach* (Line 8).

Every abstract state \hat{s} that is added to *Unreach* is a strengthening of the safety property by an additional constraint that needs to be obeyed in any composition-invariant pair, where obtaining a composition-invariant pair is the target of the algorithm. This makes our algorithm *property directed*.

If an abstract state that satisfies $\varphi_{\text{pre}}(\mathcal{B})$ is added to *Unreach*, then Algorithm 1 determines that no solution exists (Line 9). Otherwise, it generates a new constraint for E based on the abstract state preceding \hat{s} in the abstract counterexample (Line 12).

Constructing the Next Candidate Self Composition Function. Given the set of constraints in E and the formula *Unreach*, *Modify_SC* (Line 13) generates the next candidate composition function by (i) taking a constraint (\hat{s}, M) such that $\hat{s} \not\models \text{Unreach}$ (typically the one that was added last), (ii) selecting a non-starving value M_{new} for \hat{s} (such

⁶ If the conditions $\{C_M\}_M$ defining f may overlap, we consider the condition C_M by which the transition from \hat{s}_m to \hat{s}_{m+1} was defined.

a value must exist, otherwise \hat{s} would have been added to *Unreach*), and (iii) updating the conditions defining f' as follows:

$$C'_M = C_M \wedge \neg \hat{s}(\mathcal{P}) \quad C'_{M_{\text{new}}} = (C_{M_{\text{new}}} \vee \hat{s}(\mathcal{P}))$$

The conditions of other values remain as before. This definition is facilitated by the fact that the same set of predicates is used both for defining f' and for defining the abstract states $\hat{s} \in \hat{S}$ (by which *Inv* is obtained). Note that in practice we do not explicitly turn f' to be undefined for $\gamma(\text{Unreach})$. However, these definitions are ignored. The definition ensures that f' is non-starving (satisfying condition **S3**) and that no two conditions $C'_{M_1} \neq C'_{M_2}$ overlap. While the latter is not required, it also does not restrict the generality of the approach (since the language we consider is closed under Boolean operations).

Theorem 2. *Let T be a transition system, $(\text{pre}, \text{post})$ a k -safety property and \mathcal{P} a set of predicates over $\mathcal{V}^{\parallel k}$. If Algorithm 1 returns “no solution” then there is no composition-invariant pair for T and $(\text{pre}, \text{post})$ in $\mathcal{L}_{\mathcal{P}}$. Otherwise, $(f, \text{Inv}(\mathcal{P}))$ returned by Algorithm 1 is a composition-invariant pair in $\mathcal{L}_{\mathcal{P}}$, and thus $T \models^k (\text{pre}, \text{post})$.*

Complexity. Each iteration of Algorithm 1 adds at least one constraint to E , excluding a potential value for f over some abstract state \hat{s} . An excluded values is never re-used. Hence, the number of iterations is at most the number of abstract states, $2^{|\mathcal{P}|}$, multiplied by the number of potential values for each abstract state, $n = 2^k$. Altogether, the number of iterations is at most $O(2^{|\mathcal{P}|} \cdot 2^k)$. Each iteration makes one call to `Abs_Reach` which checks reachability via predicate abstraction, hence, assuming that satisfiability checks in the original logic are at most exponential, its complexity is $2^{O(|\mathcal{P}|)}$. Therefore, the overall complexity of the algorithm is $2^{O(|\mathcal{P}|)+k}$. Typically, k is a small constant, hence the complexity is dominated by $2^{O(|\mathcal{P}|)}$.

5 Evaluation and Conclusion

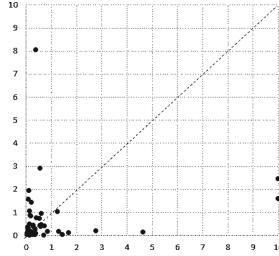
Implementation. We implemented PDSC (Algorithm 1) in Python on top of Z3 [25]. Its input is a transition system encoded by Constrained Horn Clauses (CHC) in SMT2 format, a k -safety property and a set of predicates. The abstraction is implicitly encoded using the approach of [6], and is parameterized by a composition function that is modified in each iteration. For reachability checks (`Abs_Reach`) we use SPACER [22], which supports LRA and arrays. For the set of predicates used by PDSC, we implemented an automatic procedure that mines these predicates from the CHC. Additional predicates may be added manually.

Experiments. To evaluate PDSC, we compare it to SYNONYM [26], the current state of the art in k -safety verification.

To show the effectiveness of PDSC, we consider examples that require a *nontrivial* composition (these examples are detailed in [29]). We emphasize that the motivation for these example is originated in real-life scenarios. For example, Fig. 1 follows a pattern of constant-time execution. The results of these experiments are summarized in Table 1.

Table 1. Examples that require semantic compositions

Program	PDSC		SYNONYM
	Time(s)	Iterations	
DoubleSquareNI	7	33	fail
HalfSquareNI	3.4	28	fail
ArrayIntMod	58.2	168	fail
SquaresSum	2.8	4	fail
ArrayInsert	19.5	102	fail

**Fig. 2.** Runtime comparison (in sec.): PDSC (x-axis) and SYNONYM (y-axis).

PDSC is able to find the right composition function and prove all of the examples, while SYNONYM cannot verify any of them. We emphasize that for these examples, lock-step composition is not sufficient. However, PDSC infers a composition that depends on the programs' state (variable values), rather than just program locations.

Next we consider Java programs from [26, 30], which we manually converted to C, and then converted to CHC using SEAHORN [19]. For all but 3 examples, only 2 types of predicates, which we mined automatically, were sufficient for verification: (i) relational predicates derived from the pre- and post-conditions, and (ii) for simple loops that have an index variable (e.g., for iterating over an array), an equality predicate between the copies of the indices. These predicates were sufficient since we used a large-step encoding of the transition relation, hence the abstraction via predicates takes effect only at cut-points. For the remaining 3 examples, we manually added 2–4 predicates. With the exception of 1 example where a timeout of 10 seconds was reached, all examples were solved with a lock-step composition function. Yet, we include them to show that on examples with simple compositions PDSC performs similarly to SYNONYM. This can be seen in Fig. 2.

Conclusion and Future Work. This work formulates the problem of inferring a self composition function together with an inductive invariant for the composed program, thus capturing the interplay between the self composition and the difficulty of verifying the resulting composed program. To address this problem we present PDSC— an algorithm for inferring a semantic self composition, directed at verifying the composed program with a given language of predicates. We show that PDSC manages to find non-trivial self compositions that are beyond reach of existing tools. In future work, we are interested in further improving PDSC by extending it with additional (possibly lazy) predicate discovery abilities. This has the potential to both improve performance and verify properties over wider range of programs. Additionally, we consider exploring further generalization techniques during the inference procedure.

Acknowledgements. This publication is part of a project that has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No [759102-SVIS]). The research was partially supported by Len Blavatnik and the Blavatnik Family foundation, the Blavatnik Interdisciplinary

Cyber Research Center, Tel Aviv University, the Israel Science Foundation (ISF) under grant No. 1810/18 and the United States-Israel Binational Science Foundation (BSF) grant No. 2016260.

References

1. Antonopoulos, T., Gazzillo, P., Hicks, M., Koskinen, E., Terauchi, T., Wei, S.: Decomposition instead of self-composition for proving the absence of timing channels. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18–23, 2017. pp. 362–375 (2017). <https://doi.org/10.1145/3062341.3062378>
2. Barthe, G., Crespo, J.M., Kunz, C.: Relational verification using product programs. In: Proceedings of the FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland, June 20–24, 2011, pp. 200–214 (2011). https://doi.org/10.1007/978-3-642-21437-0_17
3. Barthe, G., Crespo, J.M., Kunz, C.: Beyond 2-safety: asymmetric product programs for relational program verification. In: Artemov, S., Nerode, A. (eds.) LFCS 2013. LNCS, vol. 7734, pp. 29–43. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35722-0_3
4. Barthe, G., D’Argenio, P.R., Rezk, T.: Secure information flow by self-composition. In: 17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004), 28–30 June 2004, Pacific Grove, CA, USA. pp. 100–114 (2004). <https://doi.org/10.1109/CSFW.2004.17>
5. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 70–87. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18275-4_7
6. Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: IC3 modulo theories via implicit predicate abstraction. In: Abrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 46–61. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_4
7. Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.): Handbook of Model Checking. Springer, Cham (2018)
8. Clarkson, M.R., Schneider, F.B.: Hyperproperties. In: Proceedings of the 21st IEEE Computer Security Foundations Symposium, CSF 2008, Pittsburgh, Pennsylvania, USA, 23–25 June 2008, pp. 51–65 (2008). <https://doi.org/10.1109/CSF.2008.7>
9. Clarkson, M.R., Schneider, F.B.: Hyperproperties. J. Comput. Secur. **18**(6), 1157–1210 (2010)
10. Dahiya, M., Bansal, S.: Black-box equivalence checking across compiler optimizations. In: Chang, B.-Y.E. (ed.) APLAS 2017. LNCS, vol. 10695, pp. 127–147. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-71237-6_7
11. De Angelis, E., Fioravanti, F., Pettorossi, A., Proietti, M.: Relational verification through horn clause transformation. In: Rival, X. (ed.) SAS 2016. LNCS, vol. 9837, pp. 147–169. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53413-7_8
12. Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. Commun. ACM **20**(7), 504–513 (1977). <https://doi.org/10.1145/359636.359712>
13. Eén, N., Mishchenko, A., Brayton, R.K.: Efficient implementation of property directed reachability. In: International Conference on Formal Methods in Computer-Aided Design, FMCAD 2011, Austin, TX, USA, October 30 - November 02, 2011, pp. 125–134 (2011). <http://dl.acm.org/citation.cfm?id=2157675>
14. Eilers, M., Müller, P., Hitz, S.: Modular product programs. In: Ahmed, A. (ed.) ESOP 2018. LNCS, vol. 10801, pp. 502–529. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89884-1_18

15. Felsing, D., Grebing, S., Klebanov, V., Rümmer, P., Ulbrich, M.: Automating regression verification. In: ACM/IEEE International Conference on Automated Software Engineering, ASE 2014, Västerås, Sweden - September 15–19, 2014, pp. 349–360 (2014). <https://doi.org/10.1145/2642937.2642987>
16. Godlin, B., Strichman, O.: Regression verification. In: Proceedings of the 46th Design Automation Conference, DAC 2009, San Francisco, CA, USA, July 26–31, 2009. pp. 466–471 (2009). <https://doi.org/10.1145/1629911.1630034>
17. Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997). https://doi.org/10.1007/3-540-63166-6_10
18. Gupta, S., Saxena, A., Mahajan, A., Bansal, S.: Effective use of SMT solvers for program equivalence checking through invariant-sketching and query-decomposition. In: Beyersdorff, O., Wintersteiger, C.M. (eds.) SAT 2018. LNCS, vol. 10929, pp. 365–382. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-94144-8_22
19. Gurfinkel, A., Kahrabi, T., Komuravelli, A., Navas, J.A.: The SeaHorn verification framework. In: Kroening, D., Pasareanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 343–361. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_20
20. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
21. Karimpour, J., Isazadeh, A., Noroozi, A.A.: Verifying observational determinism. In: Federerath, H., Gollmann, D. (eds.) SEC 2015. IAICT, vol. 455, pp. 82–93. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-18467-8_6
22. Komuravelli, A., Gurfinkel, A., Chaki, S.: SMT-based model checking for recursive programs. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 17–34. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_2
23. Lahiri, S.K., Qadeer, S.: Complexity and algorithms for monomial and clausal predicate abstraction. In: Schmidt, R.A. (ed.) CADE 2009. LNCS (LNAI), vol. 5663, pp. 214–229. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02959-2_18
24. Mordvinov, D., Fedukovich, G.: Synchronizing constrained Horn clauses. In: LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, May 7–12, 2017, pp. 338–355 (2017). <http://www.easychair.org/publications/paper/340359>
25. de Moura, L., Björner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
26. Pick, L., Fedukovich, G., Gupta, A.: Exploiting synchrony and symmetry in relational verification. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 164–182. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_9
27. Saïdi, H., Shankar, N.: Abstract and model check while you prove. In: Halbwachs, N., Peled, D. (eds.) CAV 1999. LNCS, vol. 1633, pp. 443–454. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48683-6_38
28. Sharma, R., Schkufta, E., Churchill, B.R., Aiken, A.: Data-driven equivalence checking. In: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26–31, 2013. pp. 391–406 (2013). <https://doi.org/10.1145/2509136.2509509>
29. Shemer, R., Gurfinkel, A., Shoham, S., Vizel, Y.: Property directed self composition. CoRR abs/1905.07705 (2019). <http://arxiv.org/abs/1905.07705>

30. Sousa, M., Dillig, I.: Cartesian hoare logic for verifying k-safety properties. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13–17, 2016, pp. 57–69 (2016). <https://doi.org/10.1145/2908080.2908092>
31. Strichman, O., Veitsman, M.: Regression verification for unbalanced recursive functions. In: Fitzgerald, J., Heitmeyer, C., Gnesi, S., Philippou, A. (eds.) FM 2016. LNCS, vol. 9995, pp. 645–658. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48989-6_39
32. Terauchi, T., Aiken, A.: Secure information flow as a safety problem. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 352–367. Springer, Heidelberg (2005). https://doi.org/10.1007/11547662_24
33. Yang, W., Vizel, Y., Subramanyan, P., Gupta, A., Malik, S.: Lazy self-composition for security verification. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10982, pp. 136–156. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96142-2_11
34. Zaks, A., Pnueli, A.: CoVaC: compiler validation by program analysis of the cross-product. In: Cuellar, J., Maibaum, T., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 35–51. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68237-0_5

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Security-Aware Synthesis Using Delayed-Action Games

Mahmoud Elfar^(✉), Yu Wang, and Miroslav Pajic

Duke University, Durham, NC 27708, USA

{mahmoud.elfar,yu.wang094,miroslav.pajic}@duke.edu

Abstract. Stochastic multiplayer games (SMGs) have gained attention in the field of strategy synthesis for multi-agent reactive systems. However, standard SMGs are limited to modeling systems where all agents have full knowledge of the state of the game. In this paper, we introduce delayed-action games (DAGs) formalism that simulates hidden-information games (HIGs) as SMGs, where hidden information is captured by delaying a player’s actions. The elimination of private variables enables the usage of SMG off-the-shelf model checkers to implement HIGs. Furthermore, we demonstrate how a DAG can be decomposed into subgames that can be independently explored, utilizing parallel computation to reduce the model checking time, while alleviating the state space explosion problem that SMGs are notorious for. In addition, we propose a DAG-based framework for strategy synthesis and analysis. Finally, we demonstrate applicability of the DAG-based synthesis framework on a case study of a human-on-the-loop unmanned-aerial vehicle system under stealthy attacks, where the proposed framework is used to formally model, analyze and synthesize security-aware strategies for the system.

1 Introduction

Stochastic multiplayer games (SMGs) are used to model reactive systems where nondeterministic decisions are made by multiple players [4, 13, 23]. SMGs extend probabilistic automata by assigning a player to each choice to be made in the game. This extension enables modeling of complex systems where the behavior of players is unknown at design time. The *strategy synthesis* problem aims to find a *winning strategy*, i.e., a strategy that guarantees that a set of objectives (or winning conditions) is satisfied [6, 21]. Algorithms for synthesis include, for instance, value iteration and strategy iteration techniques, where multiple reward-based objectives are satisfied [2, 9, 17]. To tackle the state-space explosion problem, [29] presents an *assume-guarantee* synthesis framework that relies on synthesizing strategies on the component level first, before composing them into a global winning strategy. Mean-payoffs and ratio rewards are further investigated in [3]

This work was supported by the NSF CNS-1652544 grant, as well as the ONR N00014-17-1-2012 and N00014-17-1-2504, and AFOSR FA9550-19-1-0169 awards.

to synthesize ε -optimal strategies. Formal tools that support strategy synthesis via SMGs include PRISM-games [7, 19] and Uppaal Stratego [10].

SMGs are classified based on the number of players that can make choices at each state. In *concurrent* games, more than one player is allowed to concurrently make choices at a given state. Conversely, *turn-based* games assign one player at most to each state. Another classification considers the information available to different players across the game [27]. *Complete-information* games (also known as *perfect-information* games [5]) grant all players complete access to the information within the game. In *symmetric* games, some information is equally hidden from all players. On the contrary, *asymmetric* games allow some players to have access to more information than the others [27].

This work is motivated by security-aware systems in which stealthy adversarial actions are potentially hidden from the system, where the latter can probabilistically and intermittently gain full knowledge about the current state. While hidden-information games (HIGs) can be used to model such systems by using private variables to capture hidden information [5], standard model checkers can only synthesize strategies for (full-information) SMGs; thus, demanding for alternative representations. The equivalence between turn-based semi-perfect information games and concurrent perfect-information games was shown [5]. Since a player's strategy mainly rely on full knowledge of the game state [9], using SMGs for synthesis produces strategies that may violate synthesis specifications in cases where required information is hidden from the player. *Partially-observable* stochastic games (POSGs) allow agents to have different belief states by incorporating uncertainty about both the current state and adversarial plans [15]. Techniques such as active sensing for online replanning [14] and grid-based abstractions of belief spaces [24] were proposed to mitigate synthesis complexity arising from partial observability. The notion of *delaying actions* has been studied as means for gaining information about a game to improve future strategies [18, 30], but was not deployed as means for hiding information.

To this end, we introduce delayed-action games (DAGs)—a new class of games that simulate HIGs, where information is hidden from one player by delaying the actions of the others. The omission of private variables enables the use of off-the-shelf tools to implement and analyze DAG-based models. We show how DAGs (under some mild and practical assumptions) can be decomposed into subgames that can be independently explored, reducing the time required for synthesis by employing parallel computation. Moreover, we propose a DAG-based framework for strategy synthesis and analysis of security-aware systems. Finally, we demonstrate the framework's applicability through a case study of security-aware planning for an unmanned-aerial vehicle (UAV) system prone to stealthy cyber attacks, where we develop a DAG-based system model and further synthesize strategies with strong probabilistic security guarantees.

The paper is organized as follows. Section 2 presents SMGs, HIGs, and problem formulation. In Sect. 3, we introduce DAGs and show that they can simulate HIGs. Section 4 proposes a DAG-based synthesis framework, which we use for security-aware planning for UAVs in Sect. 5, before concluding the paper in Sect. 6.

2 Stochastic Games

In this section, we present turn-based stochastic games, which assume that all players have full information about the game state. We then introduce hidden-information games and their private-variable semantics.

Notation. We use \mathbb{N}_0 to denote the set of non-negative integers. $\mathcal{P}(A)$ denotes the powerset of A (i.e., 2^A). A variable v has a set of valuations $Ev(v)$, where $\eta(v) \in Ev(v)$ denotes one. We use Σ^* to denote the set of all finite words over alphabet Σ , including the empty word ϵ . The mapping $Eff : \Sigma^* \times Ev(v) \rightarrow Ev(v)$ indicates the effect of a finite word on $\eta(v)$. Finally, for general indexing, we use s_i or $s^{(i)}$, for $i \in \mathbb{N}_0$, while PL_γ denotes Player γ .

Turn-Based Stochastic Games (SMGs). SMGs can be used to model reactive systems that undergo both stochastic and nondeterministic transitions from one state to another. In a *turn-based* game,¹ actions can be taken at any state by at most one player. Formally, an SMG can be defined as follows [1, 28, 29].

Definition 1 (Turn-Based Stochastic Game). A turn-based game (SMG) with players $\Gamma = \{\text{I}, \text{II}, \bigcirc\}$ is a tuple $\mathcal{G} = \langle S, (S_{\text{I}}, S_{\text{II}}, S_{\bigcirc}), A, s_0, \delta \rangle$, where

- S is a finite set of states, partitioned into S_{I} , S_{II} and S_{\bigcirc} ;
- $A = A_{\text{I}} \cup A_{\text{II}} \cup \{\tau\}$ is a finite set of actions where τ is an empty action;
- $s_0 \in S_{\text{II}}$ is the initial state; and
- $\delta : S \times A \times S \rightarrow [0, 1]$ is a transition function, such that $\delta(s, a, s') \in \{1, 0\}$, $\forall s \in S_{\text{I}} \cup S_{\text{II}}, a \in A$ and $s' \in S$, and $\delta(s, \tau, s') \in [0, 1]$, $\forall s \in S_{\bigcirc}$ and $s' \in S_{\text{I}} \cup S_{\text{II}}$, where $\sum_{s' \in S_{\text{I}} \cup S_{\text{II}}} \delta(s, \tau, s') = 1$ holds.

For all $s \in S_{\text{I}} \cup S_{\text{II}}$ and $a \in A_{\text{I}} \cup A_{\text{II}}$, we write $s \xrightarrow{a} s'$ if $\delta(s, a, s') = 1$. Similarly, for all $s \in S_{\bigcirc}$ we write $s \xrightarrow{p} s'$ if s' is randomly sampled with probability $p = \delta(s, \tau, s')$.

Hidden-Information Games. SMGs assume that all players have full knowledge of the current state, and hence provide *perfect-information* models [5]. In many applications, however, this assumption may not hold. A great example are security-aware models where stealthy adversarial actions can be hidden from the system; e.g., the system may not even be aware that it is under attack. On the other hand, *hidden-information* games (HIGs) refer to games where one player does not have complete access to (or knowledge of) the current state. The notion of hidden information can be formalized with the use of *private variables* (PVs) [5]. Specifically, a game state can be encoded using variables v_T and v_B , representing the true information, which is only known to PL_{I} , and PL_{II} belief, respectively.

¹ The term *turn-based* indicates that at any state only one player can play an action. It does not necessarily imply that players take fair turns.

Definition 2 (Hidden-Information Game). A hidden-information stochastic game (HIG) with players $\Gamma = \{\text{I}, \text{II}, \bigcirc\}$ over a set of variables $V = \{v_T, v_B\}$ is a tuple $\mathcal{G}_H = \langle S, (S_I, S_{II}, S_\bigcirc), A, s_0, \beta, \delta \rangle$, where

- set of states $S \subseteq Ev(v_T) \times Ev(v_B) \times \mathcal{P}(Ev(v_T)) \times \Gamma$, partitioned in S_I, S_{II}, S_\bigcirc ;
- $A = A_I \cup A_{II} \cup \{\tau, \theta\}$ is a finite set of actions, where τ denotes an empty action, and θ is the action capturing PL_{II} attempt to reveal the true value v_T ;
- $s_0 \in S_{II}$ is the initial state;
- $\beta: A_{II} \rightarrow \mathcal{P}(A_I)$ is a function that defines the set of available PL_I actions, based on PL_{II} action; and
- $\delta: S \times A \times S \rightarrow [0, 1]$ is a transition function such that $\delta(s_I, a, s_\bigcirc) = \delta(s_\bigcirc, a, s_I) = 0$, and $\delta(s_{II}, \theta, s_\bigcirc), \delta(s_{II}, a, s_I), \delta(s_I, a, s_{II}) \in \{0, 1\}$ for all $s_I \in S_I, s_{II} \in S_{II}, s_\bigcirc \in S_\bigcirc$ and $a \in A$, where $\sum_{s' \in S_{II}} \delta(s_\bigcirc, \tau, s') = 1$.

In the above definition, δ only allows transitions s_I to s_{II} , s_{II} to s_I or s_\bigcirc , with s_{II} to s_\bigcirc conditioned by action θ , and probabilistic transitions s_\bigcirc to s_{II} . A game state can be written as $s = (t, u, \Omega, \gamma)$, but to simplify notation we use $s_\gamma(t, u, \Omega)$ instead, where $t \in Ev(v_T)$ is the true value of the game, $u \in Ev(v_B)$ is PL_{II} current belief, $\Omega \in \mathcal{P}(Ev(v_T)) \setminus \{\emptyset\}$ is PL_{II} belief space, and $\gamma \in \Gamma$ is the current player's index. When the truth is hidden from PL_{II} , the belief space Ω is the information set [27], capturing PL_{II} knowledge about the possible true values.

Example 1 (Belief vs. True Value). Our motivating example is a system that consists of a UAV and a human operator. For localization, the UAV mainly relies on a GPS sensor that can be compromised to effectively steer the UAV away from its original path. While aggressive attacks can be detected, some may remain stealthy by introducing only bounded errors at each

step [16, 20, 22, 26]. For example, Fig. 1 shows a UAV (PL_{II}) occupying zone A and flying north (N). An adversary (PL_I) can launch a stealthy attack targeting its GPS, introducing a bounded error (NE, NW) to remain stealthy. The set of stealthy actions available to the attacker depends on the preceding UAV action, which is captured by the function β , where $\beta(N) = \{NE, N, NW\}$. Being unaware of the attack, the UAV believes that it is entering zone C, while the true new location is D due to the attack (NE). Initially, $\eta(v_T) = \eta(v_B) = z_A$, and $\Omega = \{z_A\}$ as the UAV is certain it is in zone z_A . In s_2 , $\eta(v_B) = z_C$, yet $\eta(v_T) = z_D$. Although v_T is hidden, PL_{II} is aware that $\eta(v_T)$ is in $\Omega = \{z_B, z_C, z_D\}$.

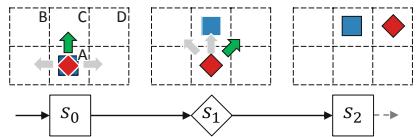


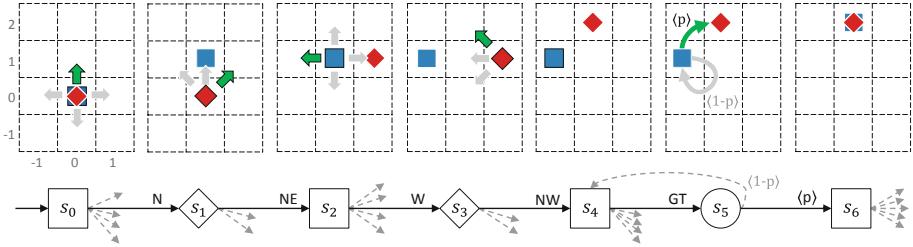
Fig. 1. The UAV belief (solid square) vs. the true value (solid diamond) of its location.

HIG Semantics. \mathcal{G}_H semantics is described using the rules shown in Fig. 2, where H2 and H3 capture PL_{II} and PL_I moves, respectively. The rule H4 specifies that a PL_{II} attempt θ to reveal the true value can succeed with probability p_i where PL_{II} belief is updated (i.e., $u' = t$), and remains unchanged otherwise.

$H1: s_0 = s_{II}(t_0, u_0, \Omega_0)$	if $t_0 = u_0, \Omega_0 = \{t_0\}$
$H2: s_{II}(t, u, \Omega) \xrightarrow{a_i} s_I(t', u', \Omega')$	if $a_i \in A_{II}, t' = t, u' = Eff(a_i, u), \Omega' = \{t' t' = Eff(b_i, t) \forall b_i \in \beta(a_i), t \in \Omega\}$
$\xrightarrow{\theta} s_{\bigcirc}(t', u', \Omega')$	if $t' = t, u' = u, \Omega' = \Omega$
$H3: s_I(t, u, \Omega) \xrightarrow{b_i} s_{II}(t', u', \Omega')$	if $b_i \in \beta(a_i), t' = Eff(b_i, t), u' = u, \Omega' = \Omega$
$H4: s_{\bigcirc}(t, u, \Omega) \xrightarrow{p_i} s_{II}(t', u', \Omega')$	if $t' = t, u' = t, \Omega' = \{t\}, p_i = \delta(s_{\bigcirc}, \tau, s_{II})$
$\xrightarrow{1-p_i} s_{II}(t', u', \Omega')$	if $t' = t, u' = u, \Omega' = \Omega, 1-p_i = \delta(s_{\bigcirc}, \tau, s_{II})$

Fig. 2. Semantic rules for an HIG.

Example 2 (HIG Semantics). Continuing Example 1, let us assume that the set of actions $A_I = A_{II} = \{N, S, E, W, NE, NW, SE, SW\}$, and that $\theta = GT$ is a geolocation task that attempts to reveal the true value of the game.² Now, consider the scenario illustrated in Fig. 3. At the initial state s_0 , the UAV attempts to move north (N), progressing the game to the state s_1 , where the adversary takes her turn by selecting an action from the set $\beta(N) = \{NE, N, NW\}$. The players take turns until the UAV performs a geolocation task GT, moving from the state s_4 to s_5 . With probability $p = \delta(s_5, \tau, s_6)$, the UAV detects its true location and updates its belief accordingly (i.e., to s_6). Otherwise, the belief remains the same (i.e., equal to s_4).

**Fig. 3.** An example of the UAV motion in a 2D-grid map, modeled as an HIG. Solid squares represent the UAV belief, while solid diamonds represent the ground truth. The UAV action GT denotes performing a geolocation task.

Problem Formulation. Following the system described in Example 2, we now consider the composed HIG $\mathcal{G}_H = \mathcal{M}_{adv} \parallel \mathcal{M}_{uav} \parallel \mathcal{M}_{as}$ shown in Fig. 4; the HIG-based model incorporates standard models of a UAV (\mathcal{M}_{uav}), an adversary (\mathcal{M}_{adv}), and a geolocation-task advisory system (\mathcal{M}_{as}) (e.g., as introduced in [11, 12]). Here, the probability of a successful detection $p(v_T, v_B)$ is a function of both the location the UAV believes to be its current location (v_B) as well

² A geolocation task is an attempt to localize the UAV by examining its camera feed.

as the ground truth location that the UAV actually occupies (v_T). Reasoning about the flight plan using such model becomes problematic since the ground truth v_T is inherently unknown to the UAV (i.e., PL_{II}), and thus so is $p(v_T, v_B)$. Furthermore, such representation, where some information is hidden, is not supported by off-the-shelf SMG model checkers. Consequently, for such HIGs, *our goal is to find an alternative representation that is suitable for strategy synthesis using off-the-shelf SMG model-checkers.*

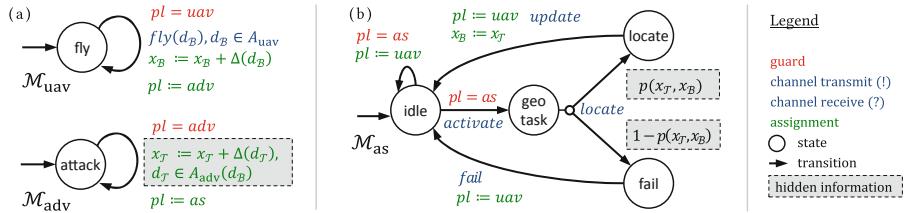


Fig. 4. An example of an HIG-based system model comprised of the UAV (\mathcal{M}_{uav}), the adversary (\mathcal{M}_{adv}), and the AS (\mathcal{M}_{as}). Framed information is hidden from the UAV-AS.

3 Delayed-Action Games

In this section, we provide an alternative representation of HIGs that eliminates the use of private variables—we introduce Delayed-Action Games (DAGs) that exploit the notion of *delayed actions*. Furthermore, we show that for any HIG, a DAG that simulates the former can be constructed.

Delayed Actions. Informally, a DAG reconstructs an HIG such that actions of PL_I (the player with access to perfect information) follow the actions of PL_{II} , i.e., PL_I actions are *delayed*. This rearrangement of the players' actions provides a means to hide information from PL_{II} without the use of private variables, since in this case, at PL_{II} states, PL_I actions have not occurred yet. In this way, PL_{II} can act as though she has complete information at the moment she makes her decision, as the future state has not yet happened and so cannot be known. In essence, the formalism can be seen as a partial ordering of the players' actions, exploiting the (partial) superposition property that a wide class of physical systems exhibit. To demonstrate this notion, let us consider DAG modeling on our running example.

Example 3 (Delaying Actions). Figure 5 depicts the (HIG-based) scenario from Fig. 3, but in the corresponding DAG, where the UAV actions are performed first (in $\hat{s}_0, \hat{s}_1, \hat{s}_2$), followed by the adversary delayed actions (in \hat{s}_3, \hat{s}_4). Note that, in the DAG model, at the time the UAV executed its actions ($\hat{s}_0, \hat{s}_1, \hat{s}_2$) the adversary actions had not occurred (yet). Moreover, \hat{s}_0 and \hat{s}_6 (Fig. 5) share the same belief and true values as s_0 and s_6 (Fig. 3), respectively, though the transient states do not exactly match. This will be used to show the relationship between the games.

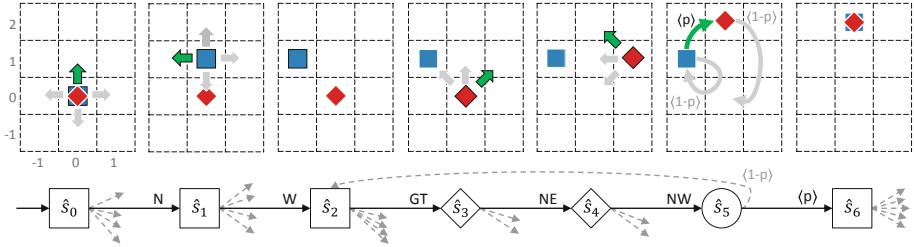


Fig. 5. The same scenario as in Fig. 3, modeled as a DAG. Solid squares represent UAV belief, while solid diamonds represent the ground truth. The UAV action GT denotes performing a geolocation task.

The advantage of this approach is twofold. First, the elimination of private variables enables simulation of an HIG using a full-information game. Thus, the formulation of the strategy synthesis problem using off-the-shelf SMG-based tools becomes feasible. In particular, a PL_{II} synthesized strategy becomes dependent on the knowledge of PL_{I} behavior (possible actions), rather than the specific (hidden) actions. We formalize a DAG as follows.

Definition 3 (Delayed-Action Game). A DAG of an HIG $\mathcal{G}_{\text{H}} = \langle S, (S_{\text{I}}, S_{\text{II}}, S_{\text{O}}), A, s_0, \beta, \delta \rangle$, with players $\Gamma = \{\text{I}, \text{II}, \text{O}\}$ over a set of variables $V = \{v_T, v_B\}$ is a tuple $\mathcal{G}_{\text{D}} = \langle \hat{S}, (\hat{S}_{\text{I}}, \hat{S}_{\text{II}}, \hat{S}_{\text{O}}), A, \hat{s}_0, \beta, \hat{\delta} \rangle$ where

- $\hat{S} \subseteq Ev(v_T) \times Ev(v_B) \times A_{\text{II}}^* \times \mathbb{N}_0 \times \Gamma$ is the set of states, partitioned into \hat{S}_{I} , \hat{S}_{II} and \hat{S}_{O} ;
- $\hat{s}_0 \in \hat{S}_{\text{II}}$ is the initial state; and
- $\hat{\delta}: \hat{S} \times A \times \hat{S} \rightarrow [0, 1]$ is a transition function such that $\hat{\delta}(\hat{s}_{\text{II}}, a, \hat{s}_{\text{O}}) = \hat{\delta}(\hat{s}_{\text{I}}, a, \hat{s}_{\text{II}}) = \hat{\delta}(\hat{s}_{\text{O}}, a, \hat{s}_{\text{I}}) = 0$, and $\hat{\delta}(\hat{s}_{\text{II}}, a, \hat{s}_{\text{II}}) \in \{0, 1\}$, $\hat{\delta}(\hat{s}_{\text{II}}, \theta, \hat{s}_{\text{I}}) \in \{0, 1\}$, $\hat{\delta}(\hat{s}_{\text{I}}, a, \hat{s}_{\text{I}}) \in \{0, 1\}$, $\hat{\delta}(\hat{s}_{\text{I}}, a, \hat{s}_{\text{O}}) \in \{0, 1\}$, for all $\hat{s}_{\text{I}} \in \hat{S}_{\text{I}}$, $\hat{s}_{\text{II}} \in \hat{S}_{\text{II}}$, $\hat{s}_{\text{O}} \in \hat{S}_{\text{O}}$ and $a \in A$, where $\sum_{\hat{s}' \in \hat{S}_{\text{II}}} \hat{\delta}(\hat{s}_{\text{O}}, a, \hat{s}') = 1$.

Note that, in contrast to transition function δ in HIG \mathcal{G}_{H} , $\hat{\delta}$ in DAG \mathcal{G}_{D} only allows transitions \hat{s}_{II} to \hat{s}_{II} or \hat{s}_{I} , as well as \hat{s}_{I} to \hat{s}_{I} or \hat{s}_{O} , and probabilistic transitions \hat{s}_{O} to \hat{s}_{II} ; also note that \hat{s}_{II} to \hat{s}_{I} is conditioned by the action θ .

DAG Semantics. A DAG state is a tuple $\hat{s} = (\hat{t}, \hat{u}, w, j, \gamma)$, which for simplicity we shorthand as $\hat{s}_{\gamma}(\hat{t}, \hat{u}, w, j)$, where $\hat{t} \in Ev(v_T)$ is the last known true value, $\hat{u} \in Ev(v_B)$ is PL_{II} belief, $w \in A_{\text{II}}^*$ captures PL_{II} actions taken since the last known true value, $j \in \mathbb{N}_0$ is an index on w , and $\gamma \in \Gamma$ is the current player index. The game transitions are defined using the semantic rules from Fig. 6. Note that PL_{II} can execute multiple moves (i.e., actions) before executing θ to attempt to reveal the true value (D2), moving to a PL_{I} state where PL_{I} executes all her delayed actions before reaching a ‘revealing’ state \hat{s}_{O} (D3). Finally, the revealing attempt can succeed with probability p_i where PL_{II} belief is updated (i.e., $\hat{u}' = \hat{t}$), or otherwise remains unchanged (D4).

- D1: $\hat{s}_0 = \hat{s}_{\text{II}}(\hat{t}_0, \hat{u}_0, w_0, 0)$ if $\hat{t}_0 = \hat{u}_0, w_0 = \epsilon$
- D2: $\hat{s}_{\text{II}}(\hat{t}, \hat{u}, w, 0) \xrightarrow{a_i} \hat{s}_{\text{II}}(\hat{t}', \hat{u}', w', 0)$ if $a_i \in A_{\text{II}}, \hat{t}' = \hat{t}, \hat{u}' = \text{Eff}(a_i, \hat{u}), w' = wa_i$
 $\xrightarrow{\theta} \hat{s}_{\text{I}}(\hat{t}', \hat{u}', w', 0)$ if $\hat{t}' = \hat{t}, \hat{u}' = \hat{u}, w' = w$
- D3: $\hat{s}_{\text{I}}(\hat{t}, \hat{u}, w, j) \xrightarrow{b_i} \hat{s}_{\text{I}}(\hat{t}', \hat{u}', w', j+1)$ if $b_i \in \beta(w_j), \hat{t}' = \text{Eff}(b_i, \hat{t}), \hat{u}' = \hat{u}, w' = w, j < |w| - 1$
 $\xrightarrow{\theta} \hat{s}_{\bigcirc}(\hat{t}', \hat{u}', w', j)$ if $b_i \in \beta(w_j), \hat{t}' = \text{Eff}(b_i, \hat{t}), \hat{u}' = \hat{u}, w' = w, j = |w| - 1$
- D4: $\hat{s}_{\bigcirc}(\hat{t}, \hat{u}, w, j) \xrightarrow{p_i} \hat{s}_{\text{II}}(\hat{t}', \hat{u}', w', 0)$ if $\hat{t}' = \hat{t}, \hat{u}' = \hat{t}, w' = \epsilon, p_i = \hat{\delta}(\hat{s}_{\bigcirc}, \hat{s}_{\text{II}})$
 $\xrightarrow{1-p_i} \hat{s}_{\text{II}}(\hat{t}', \hat{u}', w', 0)$ if $\hat{t}' = \hat{t}_0, \hat{u}' = \hat{u}, w' = w, q_i = \hat{\delta}(\hat{s}_{\bigcirc}, \hat{s}_{\text{II}})$

Fig. 6. Semantic rules for DAGs.

In both \mathcal{G}_{H} and \mathcal{G}_{D} , we label states where all players have full knowledge of the current state as *proper*. We also say that two states are similar if they agree on the belief, and equivalent if they agree on both the belief and ground truth.

Definition 4 (States). Let $s_{\gamma}(t, u, \Omega) \in S$ and $\hat{s}_{\hat{\gamma}}(\hat{t}, \hat{u}, w, j) \in \hat{S}$. We say:

- s_{γ} is proper iff $\Omega = \{t\}$, denoted by $s_{\gamma} \in \text{Prop}(\mathcal{G}_{\text{H}})$.
- $\hat{s}_{\hat{\gamma}}$ is proper iff $w = \epsilon$, denoted by $\hat{s}_{\hat{\gamma}} \in \text{Prop}(\mathcal{G}_{\text{D}})$.
- s_{γ} and $\hat{s}_{\hat{\gamma}}$ are similar iff $\hat{u} = u, \hat{t} \in \Omega$, and $\gamma = \hat{\gamma}$, denoted by $s_{\gamma} \sim \hat{s}_{\hat{\gamma}}$.
- $s_{\gamma}, \hat{s}_{\hat{\gamma}}$ are equivalent iff $t = \hat{t}, u = \hat{u}, w = \epsilon$, and $\gamma = \hat{\gamma}$, denoted by $s_{\gamma} \simeq \hat{s}_{\hat{\gamma}}$.

From the above definition, we have that $s \simeq \hat{s} \implies s \in \text{Prop}(\mathcal{G}_{\text{H}}), \hat{s} \in \text{Prop}(\mathcal{G}_{\text{D}})$. We now define *execution fragments*, possible progressions from a state to another.

Definition 5 (Execution Fragment). An execution fragment (of either an SMG, DAG or HIG) is a finite sequence of states, actions and probabilities

$$\varrho = s_0 a_1 p_1 s_1 a_2 p_2 s_2 \dots a_n p_n s_n \text{ such that } (s_i \xrightarrow{a_{i+1}} s_{i+1}) \vee (s_i \xrightarrow{\langle p_{i+1} \rangle} s_{i+1}), \forall i \geq 0. \quad ^3$$

We use *first*(ϱ) and *last*(ϱ) to refer to the first and last states of ϱ , respectively. If both states are proper, we say that ϱ is *proper* as well, denoted by $\varrho \in \text{Prop}(\mathcal{G}_{\text{H}})$.⁴ Moreover, ϱ is *deterministic* if no probabilities appear in the sequence.

Definition 6 (Move). A move m_{γ} of an execution ϱ from state $s \in \varrho$, denoted by $\text{move}_{\gamma}(s, \varrho)$, is a sequence of actions $a_1 a_2 \dots a_i \in A_{\gamma}^*$ that player γ performs in ϱ starting from s .

By omitting the player index we refer to the moves of all players. To simplify notation, we use $\text{move}(\varrho)$ as a short notation for $\text{move}(\text{first}(\varrho), \varrho)$. We write $(m)(\text{first}(\varrho)) = \text{last}(\varrho)$ to denote that the execution of move m from the *first*(ϱ) leads to the *last*(ϱ). This allows us to now define the *delay operator* as follows.

³ For deterministic transitions, $p = 1$, hence omitted from ϱ for readability.

⁴ An execution fragment lives in the transition system (TS), i.e., $\varrho \in \text{Prop}(\text{TS}(\mathcal{G}))$. We omit TS for readability.

Definition 7 (Delay Operator). For an \mathcal{G}_H , let $m = move(\varrho) = a_1 b_1 \dots a_n b_n \theta$ be a move for some deterministic $\varrho \in TS(\mathcal{G}_H)$, where $a_1 \dots a_n \in A_{II}^*$, $b_1 \dots b_n \in A_I^*$. The delay operator, denoted by \bar{m} , is defined by the rule $\bar{m} = a_1 \dots a_n \theta b_1 \dots b_n$.

Intuitively, the delay operator shifts PL_I actions to the right of PL_{II} actions up until the next probabilistic state. For example,

$$\begin{array}{ccccccccc}
\text{if } & \rho = s_{11}^{(0)} \xrightarrow{a_1} s_1^{(1)} \xrightarrow{b_2} s_{11}^{(2)} \xrightarrow{\theta} s_{\bigcirc}^{(3)} \xrightarrow{p_3} s_{11}^{(4)} \xrightarrow{a_4} s_1^{(5)} \xrightarrow{b_5} s_{11}^{(6)} \xrightarrow{a_6} s_1^{(7)} \xrightarrow{b_7} s_{11}^{(8)} \\
\text{then } & m = & a_1 & b_2 & \theta & \tau & a_4 & b_5 & a_6 & b_7, \\
\text{and } & \overline{m} = & a_1 & \theta & \cancel{b_2} & \tau & a_4 & a_6 & \cancel{b_5} & b_7.
\end{array}$$

Simulation Relation. Given an HIG \mathcal{G}_H , we first define the corresponding DAG \mathcal{G}_D .

Definition 8 (Correspondence). Given an HIG \mathcal{G}_H , a corresponding DAG $\mathcal{G}_D = \mathfrak{D}[\mathcal{G}_H]$ is a DAG that follows the semantic rules displayed in Fig. 7.

$$\begin{aligned}
s_0 = s_{11}(t_0, u_0, \Omega_0) &\implies \hat{s}_0 = \hat{s}_{11}(\hat{t}_0, \hat{u}_0, w_0, 0) \text{ s.t. } \hat{t}_0 = t_0, \hat{u}_0 = u_0 \\
s_{11}(t, u, \Omega) \xrightarrow{a_i} s_1(t', u', \Omega') &\implies \hat{s}_{11}(\hat{t}, \hat{u}, w, 0) \xrightarrow{a_i} \hat{s}_{11}(\hat{t}', \hat{u}', w', 0) \text{ s.t. } \hat{u} = u \\
s_{11}(t, u, \Omega) \xrightarrow{\theta_i} s_{\circlearrowleft}(t', u', \Omega') &\implies \hat{s}_{11}(\hat{t}, \hat{u}, w, 0) \xrightarrow{\theta_i} s_1(\hat{t}', \hat{u}', w', 0) \text{ s.t. } \hat{u} = u \\
s_1(t, u, \Omega) \xrightarrow{b_i} s_{11}(t', u', \Omega') &\implies \hat{s}_1(\hat{t}, \hat{u}, w, j) \xrightarrow{b_i} s_1(\hat{t}', \hat{u}', w', j+1) \text{ s.t. } \hat{t} = t, j < |w| \\
s_1(t, u, \Omega) \xrightarrow{b_i} s_{11}(t', u', \Omega') &\implies \hat{s}_1(\hat{t}, \hat{u}, w, j) \xrightarrow{b_i} s_{\circlearrowleft}(\hat{t}', \hat{u}', w', j) \text{ s.t. } \hat{t} = t, j = |w| \\
s_{\circlearrowleft}(t, u, \Omega) \xrightarrow{p_i} s_{11}(t', u', \Omega') &\implies \hat{s}_{\circlearrowleft}(\hat{t}, \hat{u}, w, j) \xrightarrow{p_i} \hat{s}_{11}(\hat{t}', \hat{u}', w', 0) \text{ s.t. } \hat{t} = t, \hat{u} = u \\
s_{\circlearrowleft}(t, u, \Omega) \xrightarrow{1-p_i} s_{11}(t', u', \Omega') &\implies \hat{s}_{\circlearrowleft}(\hat{t}, \hat{u}, w, j) \xrightarrow{1-p_i} \hat{s}_{11}(\hat{t}', \hat{u}', w', 0) \text{ s.t. } \hat{t} = t, \hat{u} = u
\end{aligned}$$

Fig. 7. Semantic rules for HIG-to-DAG transformation.

For the rest of this section, we consider $\mathcal{G}_D = \mathfrak{D}[\mathcal{G}_H]$, and use $\varrho \in TS(\mathcal{G}_H)$ and $\hat{\varrho} \in TS(\mathcal{G}_D)$ to denote two execution fragments of the HIG and DAG, respectively. We say that ϱ and $\hat{\varrho}$ are *similar*, denoted by $\varrho \sim \hat{\varrho}$, iff $first(\varrho) \simeq first(\hat{\varrho})$, $last(\varrho) \sim last(\hat{\varrho})$, and $move(\varrho) = move(\hat{\varrho})$.

Definition 9 (Game Proper Simulation). A game \mathcal{G}_D properly simulates \mathcal{G}_H , denoted by $\mathcal{G}_D \rightsquigarrow \mathcal{G}_H$, iff $\forall o \in \text{Prop}(\mathcal{G}_H), \exists \hat{o} \in \text{Prop}(\mathcal{G}_D)$ such that $o \sim \hat{o}$.

Before proving the existence of the simulation relation, we first show that if a move is executed on two equivalent states, then the terminal states are similar.

Lemma 1 (Terminal States Similarity). *For any $s_0 \simeq \hat{s}_0$ and a deterministic $\varrho \in \text{TS}(\mathcal{G}_{\mathbb{H}})$ where $\text{first}(\varrho) = s_0$, $\text{last}(\varrho) \in S_{\text{II}}$, then $\text{last}(\varrho) \sim \overline{\left(\text{move}(\varrho) \right)}(\hat{s}_0)$ holds.*

Proof. Let $\text{last}(\varrho_i) = s_{\gamma_i}^{(i)}(t_i, u_i, \Omega_i)$ and $\left(\overline{\text{move}(\varrho_i)}\right)(\hat{s}_0) = \hat{s}_{\gamma_i}^{(i)}(\hat{t}_i, \hat{u}_i, w_i, j_i)$, where $\text{move}(\varrho_i) = a_1 b_1 \dots a_i b_i \theta$. We then write $\text{move}(\varrho) = a_1 \dots a_i \theta b_1 \dots b_i$. We use induction over i as follows:

- Base ($i=0$): $\varrho_0 = s_0 \implies s^{(0)} \simeq \hat{s}^{(0)}$ where $u_0 = \hat{u}_0$ and $t_0 = \hat{t}_0$.
- Induction ($i > 0$): Assume that the claim holds for $\text{move}(\varrho_{i-1}) = a_1 b_1 \dots a_{i-1} b_{i-1} \theta$, i.e., $u_{i-1} = \hat{u}_{i-1}$ and $\hat{t}_{i-1} \in \Omega_{i-1}$. For ϱ_i we have that $u_i = \text{Eff}(a_i, u_{i-1})$ and $\hat{u}_i = \text{Eff}(a_i, \hat{u}_{i-1})$. Also, $t_i = \text{Eff}(b_i, t_{i-1}) \in \Omega_i$ and $\hat{t}_i = \text{Eff}(b_i, \hat{t}_{i-1})$. Hence, $u_i = \hat{u}_i$, $t_i \in \Omega_i$ and $\gamma_i = \gamma_i = \bigcirc$. Thus, $s^{(i)} \sim \hat{s}^{(i)}$ holds. The same can be shown for $\text{move}(\varrho) = a_1 b_1 \dots a_i b_i$ where no θ occurs. \square

Theorem 1 (Probabilistic Simulation). *For any $s_0 \simeq \hat{s}_0$ and $\varrho \in \text{Prop}(\mathcal{G}_H)$ where $\text{first}(\varrho) = s_0$, it holds that*

$$\Pr[\text{last}(\varrho) = s'] = \Pr\left[\left(\overline{\text{move}(\varrho)}\right)(\hat{s}_0) = \hat{s}'\right] \quad \forall s', \hat{s}' \text{ s.t. } s' \simeq \hat{s}'.$$

Proof. We can rewrite ϱ as $\varrho = \varrho_0 \xrightarrow{p_1} \varrho_1 \dots \varrho_{n-1} \xrightarrow{p_n} s_{II}^{(n)}$, where $\varrho_0, \varrho_1, \dots, \varrho_{n-1}$ are deterministic. Let $\text{first}(\varrho_i) = s_{II}^{(i)}(t_i, u_i, \Omega_i)$, $\text{last}(\varrho_i) = s_{\bigcirc}^{(i)}(t'_i, u'_i, \Omega'_i)$, and $\left(\overline{\text{move}(\varrho)}\right)(\hat{s}_0) = \hat{s}^{(n)}(\hat{t}_n, \hat{u}_n, w_n, j_n)$. We use induction over n as follows:

- Base ($n=0$): for ϱ to be deterministic and proper, $\varrho = \varrho_0 = s^{(0)}$ holds.
- Case ($n = 1$): $p_1 = p(t'_0, u'_0)$. From Lemma 1, $\hat{u}_1 = u_1$ and $\hat{t}_1 = t_1$. Hence, $\Pr[\text{last}(\varrho) = s_{II}^{(1)}] = \Pr\left[\left(\overline{\text{move}(\varrho)}\right)(\hat{s}_0) = \hat{s}_{II}^{(1)}\right] = p(t'_0, u'_0)$ and $s_{II}^{(1)} \simeq \hat{s}_{II}^{(1)}$.
- Induction ($n > 1$): It is straightforward to infer that $p_n = p(t'_{n-1}, u'_{n-1})$, hence $\Pr[\text{last}(\varrho) = s_{II}^{(n)}] = \Pr\left[\left(\overline{\text{move}(\varrho)}\right)(\hat{s}^{(0)}) = \hat{s}^{(n)}\right] = P$, and $s_{II}^{(n)} \simeq \hat{s}_{II}^{(n)}$. \square

Note that in case of multiple θ attempts, the above probability P satisfies

$$P = \prod_{i=1}^n \sum_{j=1}^{m_i} p_i(t'_{i-1}, u'_{i-1}) (1 - p_{i-1}(t'_{i-1}, u'_{i-1}))^{(j-1)},$$

where m_i is the number of θ attempts at stage i . Finally, since Theorem 1 imposes no constraints on $\text{move}(\varrho)$, a DAG can simulate all proper executions that exist in the corresponding HIG.

Theorem 2 (DAG-HIG Simulation). *For any HIG \mathcal{G}_H there exists a DAG $\mathcal{G}_D = \mathfrak{D}[\mathcal{G}_H]$ such that $\mathcal{G}_D \rightsquigarrow \mathcal{G}_H$ (as defined in Definition 9).*

4 Properties of DAG and DAG-based Synthesis

We here discuss DAG features, including how it can be decomposed into sub-games by restricting the simulation to finite executions, and the preservation of safety properties, before proposing a DAG-based synthesis framework.

Transitions. In DAGs, nondeterministic actions of different players underline different semantics. Specifically, PL_I nondeterminism captures what is known about the adversarial behavior, rather than exact actions, where PL_I actions are constrained by the earlier PL_{II} action. Conversely, PL_{II} nondeterminism abstracts the player’s decisions. This distinction reflects how DAGs can be used for strategy synthesis under hidden information. To illustrate this, suppose that a strategy π_{II} is to be obtained based on a worst-case scenario. In that case, the game is explored for all possible adversarial behaviors. Yet, if a strategy π_I is known about PL_I , a counter strategy π_{II} can be found by constructing $\mathcal{G}_D^{\pi_I}$.

Probabilistic behaviors in DAGs are captured by PL_\bigcirc , which is characterized by the transition function $\hat{\delta}: \hat{S}_\bigcirc \times \hat{S}_{II} \rightarrow [0, 1]$. The specific definition of $\hat{\delta}$ depends on the modeled system. For instance, if the transition function (i.e., the probability) is state-independent, i.e., $\hat{\delta}(\hat{s}_\bigcirc, \hat{s}_{II}) = c, c \in [0, 1]$, the obtained model becomes trivial. Yet, with a state-dependent transition function, i.e., $\hat{\delta}(\hat{s}_\bigcirc, \hat{s}_{II}) = p(\hat{t}, \hat{u})$, the probability that PL_{II} successfully reveals the true value depends on both the belief and the true value, and the transition function can then be realized since \hat{s}_\bigcirc holds both \hat{t} and \hat{u} .

Decomposition. Consider an execution $\hat{\varrho}^* = \hat{s}_0 a_1 \hat{s}_1 a_2 \hat{s}_2 \dots$ that describes a scenario where PL_{II} performs infinitely many actions with no attempt to reveal the true value. To simulate $\hat{\varrho}^*$, the word w needs to infinitely grow. Since we are interested in finite executions, we impose *stopping criteria* on the DAG, such that the game is *trapped* whenever $|w| = h_{\max}$ is true, where $h_{\max} \in \mathbb{N}$ is an *upper horizon*. We formalize the stopping criteria as a deterministic finite automaton (DFA) that, when composed with the DAG, traps the game whenever the stopping criteria hold. Note that imposing an upper horizon by itself is not a sufficient criterion for a DAG to be considered a stopping game [8]. Conversely, consider a proper (and hence finite) execution $\hat{\varrho} = \hat{s}_0 a_1 \dots \hat{s}'$, where $\hat{s}_0, \hat{s}' \in \text{Prop}(\mathcal{G}_D)$. From Definition 9, it follows that a DAG initial state is strictly proper, i.e., $\hat{s}_0 \in \text{Prop}(\mathcal{G}_D)$. Hence, when \hat{s}' is reached, the game can be seen as if it is *repeated* with a new initial state \hat{s}' . Consequently, a DAG game (complemented with stopping criteria) can be decomposed into a (possibly infinite) countable set of *subgames* that have the same structure yet different initial states.

Definition 10 (DAG Subgames). *The subgames of a \mathcal{G}_D are defined by the set $\left\{ \hat{\mathcal{G}}_i \mid \hat{\mathcal{G}}_i = \left\langle \hat{S}^{(i)}, (\hat{S}_I^{(i)}, \hat{S}_{II}^{(i)}, \hat{S}_\bigcirc^{(i)}), A, \hat{s}_0^{(i)}, \hat{\delta}^{(i)} \right\rangle, i \in \mathbb{N}_0 \right\}$, where $\hat{S} = \bigcup_i \hat{S}^{(i)}$; $\hat{S}_\gamma = \bigcup_i \hat{S}_\gamma^{(i)} \forall \gamma \in \Gamma$; and $\hat{s}_0^{(i)} = \hat{s}_{II}^{(i)}$ s.t. $\hat{s}_{II}^{(i)} \in \text{Prop}(\mathcal{G}_D^{(i)})$, $\hat{s}_{II}^{(i)} \neq \hat{s}_{II}^{(j)} \forall i, j \in \mathbb{N}_0$.*

Intuitively, each subgame either reaches a proper state (representing the initial state of another subgame) or terminates by an upper horizon. This decomposition allows for the independent (and parallel) analysis of individual subgames, drastically reducing both the time required for synthesis and the explored state space, and hence improving scalability. An example of this decompositional approach is provided in Sect. 5.

Preservation of Safety Properties. In DAGs, the action θ denotes a transition from PL_{II} to PL_I states and thus the execution of any delayed actions. While this action can simply describe a revealing attempt, it can also serve as a *what-if* analysis of how the true value may evolve at stage i of a subgame. We refer to an execution of the second type as a *hypothetical branch*, where $Hyp(\hat{\varrho}, h)$ denotes the set of hypothetical branches from $\hat{\varrho}$ at stage $h \in \{1, \dots, n\}$. Let $L_{safe}(s)$ be a labeling function denoting if a state is safe. The formula $\Phi_{safe} := [G \text{ safe}]$ is satisfied by an execution ϱ in HIG iff all $s(t, u, \Omega) \in \varrho$ are safe.

Now, consider $\hat{\varrho}$ of the DAG, with $\hat{\varrho} \sim \varrho$. We identify the following three cases:

- (a) $L_{safe}(s)$ depends only on the belief u , then $\varrho \models \Phi_{safe}$ iff all $\hat{s}_{II} \in \hat{\varrho}$ are safe;
- (b) $L_{safe}(s)$ depends only on the true value t , then $\varrho \models \Phi_{safe}$ iff all $\hat{s}_I \in Hyp(\hat{\varrho}, n)$ are safe; and
- (c) $L_{safe}(s)$ depends on both the true value t and belief u , then $\varrho \models \Phi_{safe}$ iff $last(\hat{\varrho}_h)$ is safe for all $\hat{\varrho}_h \in Hyp(\hat{\varrho}, h), h \in \{1, \dots, n\}$, where n is the number of PL_{II} actions.

Taking into account such relations, both safety (e.g., never encounter a hazard) and distance-based requirements (e.g., never exceed a subgame horizon) can be specified when using DAGs for synthesis, to ensure their satisfaction in the original model. This can be generalized to other reward-based synthesis objectives, which will be part of our future efforts that we discuss in Sect. 6.

Synthesis Framework. We here propose a framework for strategy synthesis using DAGs, which is summarized in Fig. 8. We start by formulating the automata \mathcal{M}_I , \mathcal{M}_{II} and \mathcal{M}_O , representing PL_I , PL_{II} and PL_O abstract behaviors, respectively. Next, a FIFO memory stack $(m_i)_{i=1}^n \in A_{II}^n$ is implemented using two automata \mathcal{M}_{mrd} and \mathcal{M}_{mwr} to perform reading and writing operations, respectively.⁵ The DAG \mathcal{G}_D is constructed by following Algorithm 1. The game starts with PL_{II} moves until she executes a revealing attempt θ , allowing PL_I to play her delayed actions. Once an end criterion is met, the game terminates, resembling conditions such as ‘running out of fuel’ or ‘reaching map boundaries’.

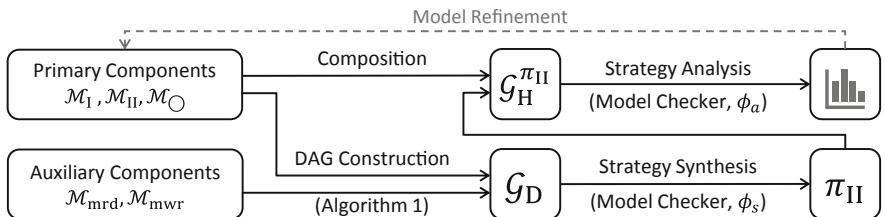


Fig. 8. Synthesis and analysis framework based on the use of DAGs.

⁵ Specific implementation details are described in Sect. 5.

Algorithm 1. Procedure for DAG construction

Input: Components $\mathcal{M}_I, \mathcal{M}_{II}, \mathcal{M}_\bigcirc, \mathcal{M}_{mwr}, \mathcal{M}_{mrd}$; initial state s_0
Result: DAG \mathcal{G}_D

```

1 while  $\neg(\text{end criterion})$  do
2   while  $a \neq \theta$  do                                 $\triangleright$  PLII plays until a revealing attempt
3      $\mathcal{M}_{II}.v_B \leftarrow \text{Eff}(a, v_B)$ ,  $\mathcal{M}_{mwr}.write(a, ++wr)$ 
4   while  $rd \leqslant wr$  do                       $\triangleright$  PLI plays all delayed actions
5      $\mathcal{M}_{mrd}.read(a, ++rd)$ ,  $\mathcal{M}_I.v_T \leftarrow \text{Eff}(\beta(a), v_T)$ 
6   if  $\text{draw } x \sim \text{Brn}(p(v_T, v_B))$  then       $\triangleright$  PL○ plays successful attempt
7      $\mathcal{M}_{II}.v_B \leftarrow \mathcal{M}_I.v_T$ ,  $wr \leftarrow 0$ ,  $rd \leftarrow 0$ 
8   else  $rd \leftarrow 0$                                  $\triangleright$  Unsuccessful attempt, forget PLI actions

```

Algorithm 2 describes the procedure for strategy synthesis based on the DAG \mathcal{G}_D , and an rPATL [6] synthesis query ϕ_{syn} that captures, for example, a safety requirement. Starting with the initial location, the procedure checks whether ϕ_{syn} is satisfied if action θ is performed at stage h , and updates the set of feasible strategies Π_i for subgame $\hat{\mathcal{G}}_i$ until h_{\max} is reached or ϕ_{syn} is not satisfied.⁶ Next, the set Π_i is used to update the list of reachable end locations ℓ with new initial locations of reachable subgames that should be explored. Finally, the composition of both \mathcal{G}_H and Π_{II}^* resolves PL_{II} nondeterminism, where the resulting model $\mathcal{G}_H^{\Pi_{II}^*}$ is a Markov Decision Process (MDP) of complete information that can be easily used for further analysis.

5 Case Study

In this section, we consider a case study where a human operator supervises a UAV prone to stealthy attacks on its GPS sensor. The UAV mission is to visit a number of targets after being airborne from a known base (initial state), while avoiding hazard zones that are known a priori. Moreover, the presence of adversarial stealthy attacks via GPS spoofing is assumed. We use the DAG framework to synthesize strategies for both the UAV and an operator advisory system (AS) that schedules geolocation tasks for the operator.

Modeling. We model the system as a delayed-action game \mathcal{G}_D , where PL_I and PL_{II} represent the adversary and the UAV-AS coalition, respectively. Figure 9 shows the model primary and auxiliary components. In the UAV model \mathcal{M}_{uav} , $x_B = (x_B, y_B)$ encodes the UAV belief, and $A_{uav} = \{N, S, E, W, NE, NW, SE, SW\}$ is the set of available movements. The AS can trigger the action *activate* to initiate a geolocation task, attempting to confirm the current location. The adversary behavior is abstracted by \mathcal{M}_{adv} where $x_T = (x_T, y_T)$ encodes the UAV true location. The adversarial actions are limited to one directional

⁶ Failing to find a strategy at stage i implies the same for all horizons of size $j > i$.

Algorithm 2. Procedure for strategy synthesis

Input: Initial location (x_0, y_0) , synthesis query ϕ_{syn}
Output: PL_{II} strategies Π_{II}^*

```

1  $\ell \leftarrow [(x_0, y_0)]$ ,  $i \leftarrow 0$ 
2 while  $i < |\ell|$  do ▷ Explore all reachable subgames
3    $\hat{s}_0 \leftarrow (\ell[i], \ell[i], \epsilon, 0, \text{II})$ ,  $h \leftarrow 1$ ,  $\text{stop} \leftarrow \perp$  ▷ Construct initial state
4   while  $h \leq h_{\max} \wedge \neg \text{stop}$  do ▷ Explore subgame till upper horizon
5      $(\pi_{\text{II}}, \varphi) \leftarrow \text{Synth}\left(\hat{\mathcal{G}}_{\hat{s}_0}^{\pi_h}, \phi_{\text{syn}}\right)$  ▷ Synthesize strategy for horizon h
6     if  $\pi_{\text{II}} \neq \emptyset$  then
7        $\Pi_i \leftarrow \Pi_i \cup (\pi_{\text{II}}, \pi_h, \varphi)$ ,  $h \leftarrow h + 1$  ▷ Save synthesized strategy
8     else  $\text{stop} \leftarrow \top$ 
9   Prune  $(\Pi_t)$ ,  $\Pi_{\text{II}}^* \leftarrow \Pi_{\text{II}}^* \cup \Pi_t$  ▷ Prune subgame strategies
10   $\ell \leftarrow \ell \cdot (\text{Reachable}(\Pi_t) \setminus \ell)$ ,  $i \leftarrow i + 1$  ▷ update reachability

```

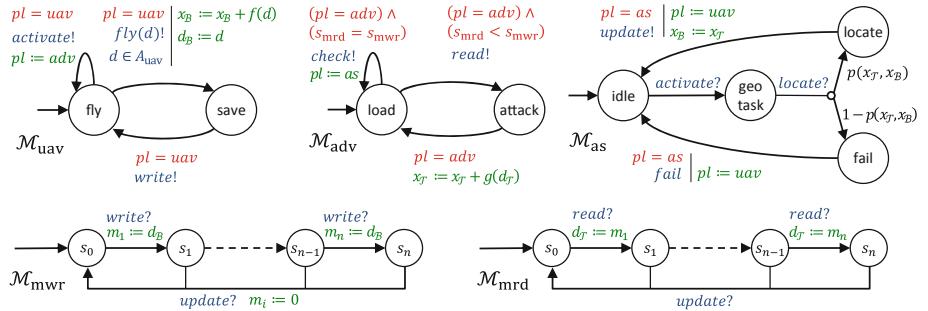


Fig. 9. Primary DAG components: UAV (\mathcal{M}_{uav}), adversary (\mathcal{M}_{adv}), and AS (\mathcal{M}_{as}). Auxiliary DAG components: memory write (\mathcal{M}_{mwr}) and memory read (\mathcal{M}_{mrd}) models, capturing the DAG representation. At stage i , the next memory location to write/read is m_i .

increment at most.⁷ If, for example, the UAV is heading N, then the adversary set of actions is $\beta(N) = \{N, NE, NW\}$. The auxiliary components \mathcal{M}_{mwr} and \mathcal{M}_{mrd} manage a FIFO memory stack $(m_i)_{i=0}^{n-1} \in A_{\text{uav}}^n$. The last UAV movement is saved in m_i by synchronizing \mathcal{M}_{mwr} with \mathcal{M}_{uav} via *write*, while \mathcal{M}_{mrd} synchronizes with \mathcal{M}_{adv} via *read* to read the next UAV action from m_j . The subgame terminates whenever action *write* is attempted and \mathcal{M}_{mwr} is at state n (i.e., out of memory).

The goal is to find strategies for the UAV-AS coalition based on the following:

- *Target reachability.* To overcome cases where targets are unreachable due to hazard zones, the label *reach* is assigned to the set of states with acceptable checkpoint locations (including the target) to render the objective incremen-

⁷ To detect aggressive attacks, techniques from literature (e.g., [16, 25, 26]) can be used.

tally feasible. The objective for all encountered subgames is then formalized as $\Pr_{\max} [\mathsf{F} \text{ reach}] \geq p_{\min}$ for some bound p_{\min} .

- *Hazard Avoidance*. Similar to target reachability, the label *hazard* is assigned to states corresponding to hazard zones. The objective $\Pr_{\max} [\mathsf{G} \neg\text{hazard}] \geq p_{\min}$ is then specified for all encountered subgames.

By refining the aforementioned objectives, synthesis queries are used for both the subgames and the supergame. Specifically, the query

$$\phi_{\text{syn}}(k) := \langle\langle \text{uav} \rangle\rangle \Pr_{\max=?} [\neg\text{hazard} \mathsf{U}^{\leq k} (\text{locate} \wedge \text{reach})] \quad (1)$$

is specified for each encountered subgame $\hat{\mathcal{G}}_i$, where *locate* indicates a successful geolocation task. By following Algorithm 2 for a q number of reachable subgames, the supergame is reduced to an MDP $\mathcal{G}_D^{\{\pi_i\}_{i=1}^q}$ (whose states are the reachable subgames), which is checked against the query

$$\phi_{\text{ana}}(n) := \langle\langle \text{adv} \rangle\rangle \Pr_{\min, \max=?} [\mathsf{F}^{\leq n} \text{target}] \quad (2)$$

to find the bounds on the probability that the target is reached under a maximum number of geolocation tasks n .

Experimental Results. Figure 10(a) shows the map setting used for implementation. The UAV’s ability to actively detect an attack depends on both its belief and the ground truth. Specifically, the probability of success in a geolocation task mainly relies on the disparity between the belief and true locations, captured by $f_{\text{dis}}: Ev(x_B) \times Ev(x_T) \rightarrow [0, 1]$, obtained by assigning probabilities for each pair of locations according to their features (e.g., landmarks) and smoothed using a Gaussian 2D filter. A thorough experimental analysis where probabilities are extracted from experiments with human operators is described in [11]. The set of hazard zones include the map boundaries to prevent the UAV from reaching boundary values. Also, the adversary is prohibited from launching attacks for at least the first step, a practical assumption to prevent the UAV model from infinitely bouncing around the target location.

We implemented the model in PRISM-games [7, 19] and performed the experiments on an Intel Core i7 4.0 GHz CPU, with 10 GB RAM dedicated to the tool. Figure 10(b) shows the supergame obtained by following the procedure in Algorithm 2. A vertex $\hat{\mathcal{G}}_{xy}$ represents a subgame (composed with its strategy) that starts at location (x, y) , while the outgoing edges points to subgames reachable from the current one. Note that each edge represents a probabilistic transition. Subgames with more than one outgoing transition imply nondeterminism that is resolved by the adversary actions. Hence, the directed graph depicts an MDP.

The synthesized strategy for $(h_{\text{adv}} = 2, h = 4)$ is demonstrated in Fig. 10(c). For the initial subgame, Fig. 11(a) shows the maximum probability of a successful geolocation task if performed at stage h , and the remaining distance to target. Assuming the adversary can launch attacks after stage $h_{\text{adv}} = 2$, the detection probability is maximized by performing the geolocation task at step 4,

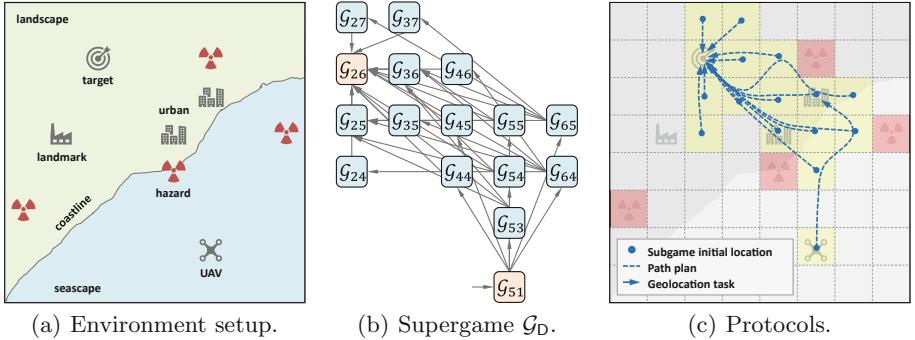


Fig. 10. (a) The environment setup used for the case study; (b) the induced supergame MDP, where the subgames form its states; and (c) the synthesized protocols.

and hazard areas can still be avoided up till $h = 6$. For $h_{\text{adv}} = 1$, however, $h = 3$ has the highest probability of success, which diminishes at $h = 6$ as no possible flight plan exists without encountering a hazard zone. The effect of the maximum number of geolocation tasks (n) on target reachability is studied by analyzing the supergame against ϕ_{ana} as shown in Fig. 11(b). The minimum number of geolocation tasks to guarantee a non-zero probability of reaching the target (regardless of the adversary strategy) is 3 with probability bounds of (33.7%, 94.4%).

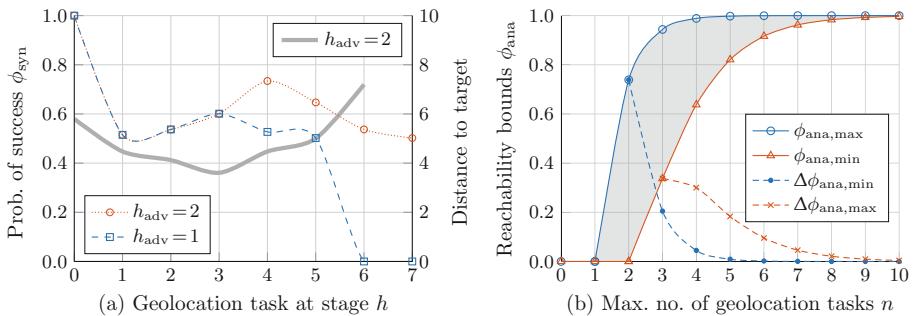


Fig. 11. Analysis results for (a) subgame $\hat{\mathcal{G}}_{51}$ and (b) supergame \mathcal{G}_D .

The experimental data obtained for this case study are listed in Table 1. For the same grid size, more complex maps require more time for synthesis while the state space size remains unaffected. The state space grows exponentially with the explored horizon size, i.e., $\mathcal{O}((|A_{\text{uav}}||A_{\text{adv}}|)^h)$, and is typically slowed by, e.g., the presence of hazard areas, since the branches of the game transitions are trimmed upon encountering such areas. Interestingly, for $h = 6$ and $h = 7$,

while the model construction time (size) for $h_{\text{adv}} = 1$ is almost twice (quadruple) as those for $h_{\text{adv}} = 2$, the time for checking ϕ_{syn} declines in comparison. This reflects the fact that, in case of $h_{\text{adv}} = 1$ compared to $h_{\text{adv}} = 2$, the UAV has higher chances to reach a hazard zone for the same k , leading to a shorter time for model checking.

Table 1. Results for strategy synthesis using queries ϕ_{syn} and ϕ_{ana} .

Subgame $\hat{\mathcal{G}}_{51}$			Model size			Time (sec)			
Map	t_{adv}	k	States	Transitions	Choices	Model	ϕ_{syn}	ϕ_{ana}	
8 × 8	1	4	11,608	17,397	15,950	2.810	0.072	–	
		5	57,129	87,865	83,267	14.729	0.602	–	
		6	236,714	366,749	359,234	62.582	1.293	–	
		7	876,550	1,365,478	1,355,932	231.741	6.021	–	
	2	4	6,678	9,230	8,394	2.381	0.042	–	
		5	33,904	48,545	45,354	10.251	0.367	–	
		6	141,622	204,551	198,640	37.192	1.839	–	
		7	524,942	763,144	754,984	145.407	8.850	–	
Supergame \mathcal{G}_D		6,212	8,306	6,660	2.216	–	2.490		

6 Discussion and Conclusion

In this paper, we introduced DAGs and showed how they can simulate HIGs by delaying players' actions. We also derived a DAG-based framework for strategy synthesis and analysis using off-the-shelf SMG model checkers. Under some practical assumptions, we showed that DAGs can be decomposed into independent subgames, utilizing parallel computation to reduce the time needed for model analysis, as well as the size of the state space. We further demonstrated the applicability of the proposed framework on a case study focused on synthesis and analysis of active attack detection strategies for UAVs prone to cyber attacks.

DAGs come at the cost of increasing the total state space size as \mathcal{M}_{mrd} and \mathcal{M}_{mwr} are introduced. This does not present a significant limitation due to the compositional approach towards strategy synthesis using subgames. However, the synthesis is still limited to model sizes that off-the-shelf tools can handle.

The concept of delaying actions implicitly assumes that the adversary knows the UAV actions *a priori*. This does not present a concern in the presented case study as an abstract (i.e., nondeterministic) adversary model is analogous to synthesizing against the worst-case attacking scenario. Nevertheless, strategies synthesized using DAGs (and SMGs in general) are inherently conservative. Depending on the considered system, this can easily lead to no feasible solution.

The proposed synthesis framework ensures preservation of safety properties. Yet, general reward-based strategy synthesis is to be approached with care. For example, rewards dependent on the belief can appear in any state, and exploring hypothetical branches is not required. However, rewards dependent on a state's true value should only appear in proper states, and all hypothetical branches are to be explored. A detailed investigation of how various properties are preserved by DAGs, along with multi-objective synthesis, is a direction for future work.

References

1. Baier, C., Brázdil, T., Grosser, M., Kucera, A.: Stochastic game logic. In: Fourth International Conference on the Quantitative Evaluation of Systems, QEST 2007, pp. 227–236. IEEE (2007). <https://doi.org/10.1109/QEST.2007.38>
2. Basset, N., Kwiatkowska, M., Topcu, U., Wiltsche, C.: Strategy synthesis for stochastic games with multiple long-run objectives. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 256–271. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_22
3. Basset, N., Kwiatkowska, M., Wiltsche, C.: Compositional strategy synthesis for stochastic games with multiple objectives. Information and Computation (2017). <https://doi.org/10.1016/j.ic.2017.09.010>
4. Brázdil, T., Chatterjee, K., Křetínský, J., Toman, V.: Strategy representation by decision trees in reactive synthesis. In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10805, pp. 385–407. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89960-2_21
5. Chatterjee, K., Henzinger, T.A.: Semiperfect-information games. In: Sarukkai, S., Sen, S. (eds.) FSTTCS 2005. LNCS, vol. 3821, pp. 1–18. Springer, Heidelberg (2005). https://doi.org/10.1007/11590156_1
6. 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). <https://doi.org/10.1007/s10703-013-0183-7>
7. Chen, T., Forejt, V., Kwiatkowska, M., Parker, D., Simaitis, A.: PRISM-games: a model checker for stochastic multi-player games. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 185–191. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_13
8. Chen, T., Forejt, V., Kwiatkowska, M., Simaitis, A., Wiltsche, C.: On stochastic games with multiple objectives. In: Chatterjee, K., Sgall, J. (eds.) MFCS 2013. LNCS, vol. 8087, pp. 266–277. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40313-2_25
9. Chen, T., Kwiatkowska, M., Simaitis, A., Wiltsche, C.: Synthesis for multi-objective stochastic games: an application to autonomous urban driving. In: Joshi, K., Siegle, M., Stoelinga, M., D'Argenio, P.R. (eds.) QEST 2013. LNCS, vol. 8054, pp. 322–337. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40196-1_28
10. David, A., Jensen, P.G., Larsen, K.G., Mikulčionis, M., Taankvist, J.H.: UPPAAL STRATEGO. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 206–211. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_16

11. Elfar, M., Zhu, H., Cummings, M.L., Pajic, M.: Security-aware synthesis of human-UAV protocols. In: Proceedings of 2019 IEEE International Conference on Robotics and Automation (ICRA). IEEE (2019)
12. Feng, L., Wiltsche, C., Humphrey, L., Topcu, U.: Synthesis of human-in-the-loop control protocols for autonomous systems. *IEEE Trans. Autom. Sci. Eng.* **13**(2), 450–462 (2016). <https://doi.org/10.1109/TASE.2016.2530623>
13. Fremont, D.J., Seshia, S.A.: Reactive control improvisation. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 307–326. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_17
14. Fu, J., Topcu, U.: Integrating active sensing into reactive synthesis with temporal logic constraints under partial observations. In: 2015 American Control Conference (ACC), pp. 2408–2413. IEEE (2015). <https://doi.org/10.1109/ACC.2015.7171093>
15. Hansen, E.A., Bernstein, D.S., Zilberstein, S.: Dynamic programming for partially observable stochastic games. *AAAI* **4**, 709–715 (2004)
16. Jovanov, I., Pajic, M.: Relaxing integrity requirements for attack-resilient cyber-physical systems. *IEEE Trans. Autom. Control* (2019). <https://doi.org/10.1109/TAC.2019.2898510>
17. Kelmendi, E., Krämer, J., Křetínský, J., Weininger, M.: Value iteration for simple stochastic games: stopping criterion and learning algorithm. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 623–642. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_36
18. Klein, F., Zimmermann, M.: How much lookahead is needed to win infinite games? In: Halldórsson, M.M., Iwama, K., Kobayashi, N., Speckmann, B. (eds.) ICALP 2015. LNCS, vol. 9135, pp. 452–463. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-47666-6_36
19. Kwiatkowska, M., Parker, D., Wiltsche, C.: Prism-games: verification and strategy synthesis for stochastic multi-player games with multiple objectives. *Int. J. Softw. Tools Technol. Transf.* **20**(2), 195–210 (2018)
20. Lesi, V., Jovanov, I., Pajic, M.: Security-aware scheduling of embedded control tasks. *ACM Trans. Embed. Comput. Syst. (TECS)* **16**(5s), 188:1–188:21 (2017). <https://doi.org/10.1145/3126518>
21. Li, W., Sadigh, D., Sastry, S.S., Seshia, S.A.: Synthesis for human-in-the-loop control systems. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 470–484. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_40
22. Mo, Y., Sinopoli, B.: On the performance degradation of cyber-physical systems under stealthy integrity attacks. *IEEE Trans. Autom. Control* **61**(9), 2618–2624 (2016). <https://doi.org/10.1109/TAC.2015.2498708>
23. Neider, D., Topcu, U.: An automaton learning approach to solving safety games over infinite graphs. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 204–221. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_12
24. Norman, G., Parker, D., Zou, X.: Verification and control of partially observable probabilistic real-time systems. In: Sankaranarayanan, S., Vicario, E. (eds.) FORMATS 2015. LNCS, vol. 9268, pp. 240–255. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-22975-1_16
25. Pajic, M., Lee, I., Pappas, G.J.: Attack-resilient state estimation for noisy dynamical systems. *IEEE Trans. Control Netw. Syst.* **4**(1), 82–92 (2017). <https://doi.org/10.1109/TCNS.2016.2607420>

26. Pajic, M., Weimer, J., Bezzo, N., Sokolsky, O., Pappas, G.J., Lee, I.: Design and implementation of attack-resilient cyberphysical systems: with a focus on attack-resilient state estimators. *IEEE Control Syst.* **37**(2), 66–81 (2017). <https://doi.org/10.1109/MCS.2016.2643239>
27. Rasmusen, E., Blackwell, B.: Games and Information, vol. 15. MIT Press, Cambridge (1994)
28. Svoreňová, M., Kwiatkowska, M.: Quantitative verification and strategy synthesis for stochastic games. *Eur. J. Control* **30**, 15–30 (2016). <https://doi.org/10.1016/j.ejcon.2016.04.009>
29. Wiltsche, C.: Assume-guarantee strategy synthesis for stochastic games. Ph.D. thesis, Ph.D. dissertation, Department of Computer Science, University of Oxford (2015)
30. Zimmermann, M.: Delay games with WMSO+ U winning conditions. *RAIRO Theor. Inform. Appl.* **50**(2), 145–165 (2016). <https://doi.org/10.1051/ita/2016018>

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Automated Hypersafety Verification

Azadeh Farzan^(✉) and Anthony Vandikas

University of Toronto, Toronto, Canada

azadeh@cs.toronto.edu

Abstract. We propose an automated verification technique for hypersafety properties, which express sets of valid interrelations between multiple finite runs of a program. The key observation is that constructing a proof for a small representative set of the runs of the product program (i.e. the product of the several copies of the program by itself), called a *reduction*, is sufficient to formally prove the hypersafety property about the program. We propose an algorithm based on a counterexample-guided refinement loop that simultaneously searches for a reduction and a proof of the correctness for the reduction. We demonstrate that our tool WEAVER is very effective in verifying a diverse array of hypersafety properties for a diverse class of input programs.

1 Introduction

A hypersafety property describes the set of valid interrelations between multiple finite runs of a program. A k -safety property [7] is a program safety property whose violation is witnessed by at least k finite runs of a program. Determinism is an example of such a property: non-determinism can only be witnessed by two runs of the program on the same input which produce two different outputs. This makes determinism an instance of a 2-safety property.

The vast majority of existing program verification methodologies are geared towards verifying standard (1-)safety properties. This paper proposes an approach to automatically reduce verification of k -safety to verification of 1-safety, and hence a way to leverage existing safety verification techniques for hypersafety verification. The most straightforward way to do this is via *self-composition* [5], where verification is performed on k memory-disjoint copies of the program, sequentially composed one after another. Unfortunately, the proofs in these cases are often very verbose, since the full functionality of each copy has to be captured by the proof. Moreover, when it comes to automated verification, the invariants required to verify such programs are often well beyond the capabilities of modern solvers [26] even for very simple programs and properties.

The more practical approach, which is typically used in manual or automated proofs of such properties, is to compose k memory-disjoint copies of the program *in parallel* (instead of in sequence), and then verify some *reduced* program obtained by removing redundant traces from the program formed in the previous step. This parallel product program can have many such reductions.

For example, the program formed from sequential self-composition is one such reduction of the parallel product program. Therefore, care must be taken to choose a “good” reduction that *admits a simple proof*. Many existing approaches limit themselves to a narrow class of reductions, such as the one where each copy of the program executes in lockstep [3, 10, 24], or define a general class of reductions, but do not provide algorithms with guarantees of covering the entire class [4, 24].

We propose a solution that combines the search for a safety proof with the search for an appropriate reduction, in a counterexample-based refinement loop. Instead of settling on a single reduction in advance, we try to verify the entire (possibly infinite) set of reductions simultaneously and terminate as soon as some reduction is successfully verified. If the proof is not currently strong enough to cover at least one of the represented program reductions, then an appropriate set of counterexamples are generated that guarantee progress towards a proof.

Our solution is language-theoretic. We propose a way to represent sets of reductions using infinite tree automata. The standard safety proofs are also represented using the same automata, which have the desired closure properties. This allows us to check if a candidate proof is in fact a proof for one of the represented program reductions, with reasonable efficiency.

Our approach is not uniquely applicable to hypersafety properties of sequential programs. Our proposed set of reductions naturally work well for concurrent programs, and can be viewed in the spirit of reduction-based methods such as those proposed in [11, 21]. This makes our approach particularly appealing when it comes to verification of hypersafety properties of concurrent programs, for example, proving that a concurrent program is deterministic. The parallel composition for hypersafety verification mentioned above and the parallel composition of threads inside the multi-threaded program are treated in a uniform way by our proof construction and checking algorithms. In summary:

- We present a counterexample-guided refinement loop that simultaneously searches for a proof and a program reduction in Sect. 7. This refinement loop relies on an efficient algorithm for proof checking based on the antichain method of [8], and strong theoretical progress guarantees.
- We propose an automata-based approach to representing a class of program reductions for k-safety verification. In Sect. 5 we describe the precise class of automata we use and show how their use leads to an effective proof checking algorithm incorporated in our refinement loop.
- We demonstrate the efficacy of our approach in proving hypersafety properties of sequential and concurrent benchmarks in Sect. 8.

2 Illustrative Example

We use a simple program MULT, that computes the product of two non-negative integers, to illustrate the challenges of verifying hypersafety properties and the type of proof that our approach targets. Consider the multiplication program in Fig. 1(i), and assume we want to prove that it is distributive over addition.

Mult:	Copy 1:	Copy 2:	Copy 3:
$\ell_1: i \leftarrow 0$	$\ell_1: i_1 \leftarrow 0$	$\ell_1: i_2 \leftarrow 0$	$\ell_1: i_3 \leftarrow 0$
$\ell_2: x \leftarrow 0$	$\ell_2: x_1 \leftarrow 0$	$\ell_2: x_2 \leftarrow 0$	$\ell_2: x_3 \leftarrow 0$
$\ell_3: \text{while } i < a$	$\ell_3: \text{while } i_1 < a + b$	$\ell_3: \text{while } i_2 < a$	$\ell_3: \text{while } i_3 < b$
$\ell_4: \quad \quad x \leftarrow x + b$	$\ell_4: \quad \quad x_1 \leftarrow x_1 + c$	$\ell_4: \quad \quad x_2 \leftarrow x_2 + c$	$\ell_4: \quad \quad x_3 \leftarrow x_3 + c$
$\ell_5: \quad \quad i \leftarrow i + 1$	$\ell_5: \quad \quad i_1 \leftarrow i_1 + 1$	$\ell_5: \quad \quad i_2 \leftarrow i_2 + 1$	$\ell_5: \quad \quad i_3 \leftarrow i_3 + 1$
(i)	$\ell_6:$	$\ell_6:$	(ii)

Fig. 1. Program MULT (i) and the parallel composition of three copies of it (ii).

In Fig. 1(ii), the parallel composition of MULT with two copies of itself is illustrated. The product program is formed for the purpose of proving distributivity, which can be encoded through the postcondition $x_1 = x_2 + x_3$. Since a , b , and c are not modified in the program, the same variables are used across all copies. One way to prove MULT is distributive is to come up with an inductive invariant ϕ_{ijk} for each location in the product program, represented by a triple of program locations (ℓ_i, ℓ_j, ℓ_k) , such that $true \implies \phi_{111} \text{ and } \phi_{666} \implies x_1 = x_2 + x_3$. The main difficulty lies in finding assignments for locations such as ϕ_{611} that are points in the execution of the program where one thread has finished executing and the next one is starting. For example, at (ℓ_6, ℓ_1, ℓ_1) we need the assignment $\phi_{611} \leftarrow x_1 = (a + b) * c$ which is non-linear. However, the program given in Fig. 1(ii) can be verified with simpler (linear) reasoning.

The program on the right is a semantically equivalent *reduction* of the full composition of Fig. 1(ii). Consider the program $P = (\text{Copy 1} \parallel (\text{Copy 2}; \text{Copy 3}))$. The program on the right is equivalent to a lockstep execution of the two parallel components of P . The validity of this reduction is derived from the fact that the statements in each thread are *independent* of the statements in the other. That is, reordering the statements of different threads in an execution leads to an equivalent execution. It is easy to see that $x_1 = x_2 + x_3$ is an invariant of both while loops in the reduced program, and therefore, linear reasoning is sufficient to prove the postcondition for this program. Conceptually, this reduction (and its soundness proof) together with the proof of correctness for the reduced program constitute a proof that the original program MULT is distributive. Our proposed approach can come up with reductions like this and their corresponding proofs fully automatically. Note that a lockstep reduction of the program in Fig. 1(ii) would not yield a solution for this problem and therefore the discovery of the right reduction is an integral part of the solution.

```
i1 ← 0, i2 ← 0, i3 ← 0
x1 ← 0, x2 ← 0, x3 ← 0
while i2 < a
    x1 ← x1 + c
    x2 ← x2 + c
    i1 ← i1 + 1
    i2 ← i2 + 1
while i3 < b
    x1 ← x1 + c
    x3 ← x3 + c
    i1 ← i1 + 1
    i3 ← i3 + 1
```

3 Programs and Proofs

A non-deterministic finite automaton (NFA) is a tuple $A = (Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of states, Σ is a finite alphabet, $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of final states. A deterministic finite automaton (DFA) is an NFA whose transition relation is a function $\delta : Q \times \Sigma \rightarrow Q$. The language of an NFA or DFA A is denoted $\mathcal{L}(A)$, which is defined in the standard way [18].

3.1 Program Traces

\mathcal{St} denotes the (possibly infinite) set of *program states*. For example, a program with two integer variables has $\mathcal{St} = \mathbb{Z} \times \mathbb{Z}$. $\mathcal{A} \subseteq \mathcal{St}$ is a (possibly infinite) set of *assertions* on program states. Σ denotes a finite alphabet of program *statements*. We refer to a finite string of statements as a (program) *trace*. For each statement $a \in \Sigma$ we associate a *semantics* $\llbracket a \rrbracket \subseteq \mathcal{St} \times \mathcal{St}$ and extend $\llbracket - \rrbracket$ to traces via (relation) composition. A trace $x \in \Sigma^*$ is said to be *infeasible* if $\llbracket x \rrbracket(\mathcal{St}) = \emptyset$, where $\llbracket x \rrbracket(\mathcal{St})$ denotes the image of $\llbracket x \rrbracket$ under \mathcal{St} . To abstract away from a particular program syntax, we define a *program* as a regular language of traces. The semantics of a program P is simply the union of the semantics of its traces $\llbracket P \rrbracket = \bigcup_{x \in P} \llbracket x \rrbracket$. Concretely, one may obtain programs as languages by interpreting their edge-labelled control-flow graphs as DFAs: each vertex in the control flow graph is a state, and each edge in the control flow graph is a transition. The control flow graph entry location is the initial state of the DFA and all its exit locations are final states.

3.2 Safety

There are many equivalent notions of program safety; we use non-reachability. A program P is *safe* if all traces of P are infeasible, i.e. $\llbracket P \rrbracket(\mathcal{St}) = \emptyset$. Standard partial correctness specifications are then represented via a simple encoding. Given a precondition ϕ and a postcondition ψ , the validity of the Hoare-triple $\{\phi\}P\{\psi\}$ is equivalent to the safety of $[\phi] \cdot P \cdot [\neg\psi]$, where $[\]$ is a standard assume statement (or the singleton set containing it), and \cdot is language concatenation.

Example 3.1. We use determinism as an example of how k -safety can be encoded in the framework defined thus far. If P is a program then determinism of P is equivalent to safety of $[\phi] \cdot (P_1 \sqcup P_2) \cdot [\neg\phi]$ where P_1 and P_2 are copies of P operating on disjoint variables, \sqcup is a shuffle product of two languages, and $[\phi]$ is an assume statement asserting that the variables in each copy of P are equal.

A *proof* is a finite set of assertions $\Pi \subseteq \mathcal{A}$ that includes *true* and *false*. Each Π gives rise to an NFA $\Pi_{NFA} = (\Pi, \mathcal{St}, \delta_\Pi, \text{true}, \{\text{false}\})$ where $\delta_\Pi(\phi_{pre}, a) = \{\phi_{post} \mid \llbracket a \rrbracket(\phi_{pre}) \subseteq \phi_{post}\}$. We abbreviate $\mathcal{L}(\Pi_{NFA})$ as $\mathcal{L}(\Pi)$. Intuitively, $\mathcal{L}(\Pi)$

consists of all traces that can be proven infeasible using only assertions in Π . Thus the following proof rule is sound [12, 13, 17]:

$$\frac{\exists \Pi \subseteq \mathcal{A}. P \subseteq \mathcal{L}(\Pi)}{P \text{ is safe}} \quad (\text{SAFE})$$

When $P \subseteq \mathcal{L}(\Pi)$, we say that Π is a proof for P . A proof does not uniquely belong to any particular program; a single Π may prove many programs correct.

4 Reductions

The set of assertions used for a proof is usually determined by a particular language of assertions, and a safe program may not have a (safety) proof in that particular language. Yet, a subset of the program traces may have a proof in that assertion language. If it can be proven that the subset of program runs that have a safety proof are a faithful representation of all program behaviours (with respect to a given property), then the program is correct. This motivates the notion of *program reductions*.

Definition 4.1 (semantic reduction). *If for programs P and P' , P' is safe implies that P is safe, then P' is a semantic reduction of P (written $P' \preceq P$).*

The definition immediately gives rise to the following proof rule for proving program safety:

$$\frac{\exists P' \preceq P, \Pi \subseteq \mathcal{A}. P' \subseteq \mathcal{L}(\Pi)}{P \text{ is safe}} \quad (\text{SAFERED1})$$

This generic proof rule is not automatable since, given a proof Π , verifying the existence of the appropriate reduction is *undecidable*. Observe that a program is safe if and only if \emptyset is a valid reduction of the program. This means that discovering a semantic reduction and proving safety are mutually reducible to each other. To have decidable premises for the proof rule, we need to formulate an easier (than proving safety) problem in discovering a reduction. One way to achieve this is by restricting the set of reductions under consideration from all reductions (given in Definition 4.1) to a proper subset which more amenable to algorithmic checking. Fixing a set \mathcal{R} of (semantic) reductions, we will have the rule:

$$\frac{\exists P' \in \mathcal{R}. P' \subseteq \mathcal{L}(\Pi) \quad \forall P' \in \mathcal{R}. P' \preceq P}{P \text{ is safe}} \quad (\text{SAFERED2})$$

Proposition 4.2. *The proof rule SAFERED2 is sound.*

The core contribution of this paper is that it provides an algorithmic solution inspired by the above proof rule. To achieve this, two subproblems are solved: (1) Given a set \mathcal{R} of reductions of a program P and a candidate proof Π , can we check if there exists a reduction $P' \in \mathcal{R}$ which is covered by the proof Π ? In Sect. 5, we propose a new semantic interpretation of an existing notion of infinite tree automata that gives rise to an algorithmic check for this step. (2) Given a program P , is there a general sound set of reductions \mathcal{R} that be effectively represented to accommodate step (1)? In Sect. 6, we propose a construction of an effective set of reductions, representable by our infinite tree automata, using inspirations from existing partial order reduction techniques [15].

5 Proof Checking

Given a set of reductions \mathcal{R} of a program P , and a candidate proof Π , we want to check if there exists a reduction $P' \in \mathcal{R}$ which is covered by Π . We call this *proof checking*. We use tree automata to represent certain classes of languages (i.e sets of sets of strings), and then use operations on these automata for the purpose of proof checking.

The set Σ^* can be represented as an infinite tree. Each $x \in \Sigma^*$ defines a path to a unique node in the tree: the root node is located at the empty string ϵ , and for all $a \in \Sigma$, the node located at xa is a child of the node located at x . Each node is then identified by the string labeling the path leading to it. A language $L \subseteq \Sigma^*$ (equivalently, $L : \Sigma^* \rightarrow \mathbb{B}$) can consequently be represented as an infinite tree where the node at each x is labelled with a boolean value $B \equiv (x \in L)$. An example is given in Fig. 2.

It follows that a set of languages is a set of infinite trees, which can be represented using automata over infinite trees. Looping Tree Automata (LTAs) are a subclass of Büchi Tree Automata where all states are accept states [2]. The class of Looping Tree Automata is closed under intersection and union, and checking emptiness of LTAs is decidable. Unlike Büchi Tree Automata, emptiness can be decided in linear time [2].

Definition 5.1. A Looping Tree Automaton (LTA) over $|\Sigma|$ -ary, \mathbb{B} -labelled trees is a tuple $M = (Q, \Delta, q_0)$ where Q is a finite set of states, $\Delta \subseteq Q \times \mathbb{B} \times (\Sigma \rightarrow Q)$ is the transition relation, and q_0 is the initial state.

Intuitively, an LTA $M = (Q, \Delta, q_0)$ performs a parallel and depth-first traversal of an infinite tree L while maintaining some local state. Execution begins at the root ϵ from state q_0 and non-deterministically picks a transition $(q_0, B, \sigma) \in \Delta$ such that B matches the label at the root of the tree (i.e. $B = (\epsilon \in L)$). If no such transition exists, the tree is rejected. Otherwise, M recursively works on

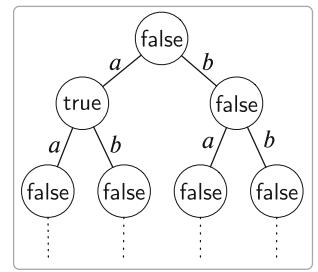


Fig. 2. Language $\{a\}$ as an infinite tree.

each child a from state $q' = \sigma(a)$ in parallel. This process continues infinitely, and L is accepted if and only if L is never rejected.

Formally, M 's execution over a tree L is characterized by a *run* $\delta^* : \Sigma^* \rightarrow Q$ where $\delta^*(\epsilon) = q_0$ and $(\delta^*(x), x \in L, \lambda a. \delta^*(xa)) \in \Delta$ for all $x \in \Sigma^*$. The set of languages accepted by M is then defined as $\mathcal{L}(M) = \{L \mid \exists \delta^*. \delta^* \text{ is a run of } M \text{ on } L\}$.

Theorem 5.2. *Given an LTA M and a regular language L , it is decidable whether $\exists P \in \mathcal{L}(M). P \subseteq L$.*

The proof, which appears in [14], reduces the problem to deciding whether $\mathcal{L}(M) \cap \mathcal{P}(L) \neq \emptyset$. LTAs are closed under intersection and have decidable emptiness checks, and the lemma below is the last piece of the puzzle.

Lemma 5.3. *If L is a regular language, then $\mathcal{P}(L)$ is recognized by an LTA.*

Counterexamples. Theorem 5.2 effectively states that proof checking is decidable. For automated verification, beyond checking the validity of a proof, we require counterexamples to fuel the development of the proof when the proof does not check. Note that in the simple case of the proof rule **SAFE**, when $P \not\subseteq \mathcal{L}(\Pi)$ there exists a counterexample trace $x \in P$ such that $x \notin \mathcal{L}(\Pi)$.

With our proof rule **SAFERED2**, things get a bit more complicated. First, note that unlike the classic case (**SAFE**), where a failed proof check coincides with the non-emptiness of an intersection check (i.e. $P \cap \overline{\mathcal{L}(\Pi)} \neq \emptyset$), in our case, a failed proof check coincides with the emptiness of an intersection check (i.e. $\mathcal{R} \cap \mathcal{P}(\mathcal{L}(\Pi)) = \emptyset$). The sets \mathcal{R} and $\mathcal{P}(\mathcal{L}(\Pi))$ are both sets of languages. What does the witness to the emptiness of the intersection look like? Each language member of \mathcal{R} contains at least one string that does not belong to any of the subsets of our proof language. One can collect all such witness strings to guarantee progress across the board in the next round. However, since LTAs can represent an infinite set of languages, one must take care not end up with an infinite set of counterexamples following this strategy. Fortunately, this will not be the case.

Theorem 5.4. *Let M be an LTA and let L be a regular language such that $P \not\subseteq L$ for all $P \in \mathcal{L}(M)$. There exists a finite set of counterexamples C such that, for all $P \in \mathcal{L}(M)$, there exists some $x \in C$ such that $x \in P$ and $x \notin L$.*

The proof appears in [14]. This theorem justifies our choice of using LTAs instead of more expressive formalisms such as Büchi Tree Automata. For example, the Büchi Tree Automaton that accepts the language $\{\{x\} \mid x \in \Sigma^*\}$ would give rise to an infinite number of counterexamples with respect to the empty proof (i.e. $\Pi = \emptyset$). The finiteness of the counterexample set presents an alternate proof that LTAs are strictly less expressive than Büchi Tree Automata [27].

6 Sleep Set Reductions

We have established so far that (1) a set of assertions gives rise to a regular language proof, and (2) given a regular language proof and a set of program reductions recognizable by an LTA, we can check the program (reductions) against the proof. The last piece of the puzzle is to show that a useful class of program reductions can be expressed using LTAs.

Recall our example from Sect. 2. The reduction we obtain is sound because, for every trace in the full parallel-composition program, an equivalent trace exists in the reduced program. By equivalent, we mean that one trace can be obtained from the other by swapping independent statements. Such an equivalence is the essence of the theory of Mazurkiewicz traces [9].

We fix a reflexive symmetric *dependence relation* $D \subseteq \Sigma \times \Sigma$. For all $a, b \in \Sigma$, we say that a and b are *dependent* if $(a, b) \in D$, and say they are *independent* otherwise. We define \sim_D as the smallest congruence satisfying $xaby \sim_D xbay$ for all $x, y \in \Sigma^*$ and independent $a, b \in \Sigma$. The closure of a language $L \subseteq \Sigma^*$ with respect to \sim_D is denoted $[L]_D$. A language L is \sim_D -closed if $L = [L]_D$. It is worthwhile to note that all input programs considered in this paper correspond to regular languages that are \sim_D -closed.

An equivalence class of \sim_D is typically called a (Mazurkiewicz) trace. We avoid using this terminology as it conflicts with our definition of traces as strings of statements in Sect. 3.1. We assume D is *sound*, i.e. $\llbracket ab \rrbracket = \llbracket ba \rrbracket$ for all independent $a, b \in \Sigma$.

Definition 6.1 (D-reduction). *A program P' is a D-reduction of a program P , that is $P' \preceq_D P$, if $[P']_D = P$.*

Note that the equivalence relation on programs induced by \sim_D is a refinement of the semantic equivalence relation used in Definition 4.1.

Lemma 6.2. *If $P' \preceq_D P$ then $P' \preceq P$.*

Ideally, we would like to define an LTA that accepts all D -reductions of a program P , but unfortunately this is not possible in general.

Proposition 6.3 (corollary of Theorem 67 of [9]). *For arbitrary regular languages $L_1, L_2 \in \Sigma^*$ and relation D , the proposition $\exists L \preceq_D L_1. L \subseteq L_2$ is undecidable.*

The proposition is decidable only when \overline{D} is transitive, which does not hold for a semantically correct notion of independence for a parallel program encoding a k -safety property, since statements from the same thread are dependent and statements from different program copies are independent. Therefore, we have:

Proposition 6.4. *Assume P is a \sim_D -closed program and Π is a proof. The proposition $\exists P' \preceq_D P. P' \subseteq \mathcal{L}(\Pi)$ is undecidable.*

In order to have a decidable premise for proof rule SAFERED2 then, we present an approximation of the set of D -reductions, inspired by sleep sets [15]. The idea is to construct an LTA that recognizes *a class of D -reductions* of an input program P , whose language is assumed to be \sim_D -closed. This automaton intuitively makes non-deterministic choices about what program traces to *prune* in favour of other \sim_D -equivalent program traces for a given reduction. Different non-deterministic choices lead to different D -reductions.

Consider two statements $a, b \in \Sigma$ where $(a, b) \notin D$. Let $x, y \in \Sigma^*$ and consider two program runs $xaby$ and $xbay$. We know $\llbracket xbay \rrbracket = \llbracket xaby \rrbracket$. If the automaton makes a non-deterministic choice that the successors of xa have been explored, then the successors of xba need not be explored (can be pruned away) as illustrated in Fig. 3. Now assume $(a, c) \in D$, for some $c \in \Sigma$. When the node xbc is being explored, we can no longer safely ignore a -transitions, since the equality $\llbracket xbcay \rrbracket = \llbracket xabcy \rrbracket$ is not guaranteed. Therefore, the a successor of xbc has to be explored. The nondeterministic choice of what child node to explore is modelled by a choice of order in which we explore each node's children. Different orders yield different reductions. Reductions are therefore characterized as an assignment $R : \Sigma^* \rightarrow \text{Lin}(\Sigma)$ from nodes to linear orderings on Σ , where $(a, b) \in R(x)$ means we explore child xa after child xb .

Given $R : \Sigma^* \rightarrow \text{Lin}(\Sigma)$, the *sleep set* $\text{sleep}_R(x) \subseteq \Sigma$ at node $x \in \Sigma^*$ defines the set of transitions that can be ignored at x :

$$\text{sleep}_R(\epsilon) = \emptyset \quad (1)$$

$$\text{sleep}_R(xa) = (\text{sleep}_R(x) \cup R(x)(a)) \setminus D(a) \quad (2)$$

Intuitively, (1) no transition can be ignored at the root node, since nothing has been explored yet, and (2) at node x , the sleep set of xa is obtained by adding the transitions we explored before a ($R(x)(a)$) and then removing the ones that conflict with a (i.e. are related to a by D). Next, we define the nodes that are ignored. The set of ignored nodes is the smallest set $\text{ignore}_R : \Sigma^* \rightarrow \mathbb{B}$ such that

$$x \in \text{ignore}_R \implies xa \in \text{ignore}_R \quad (1)$$

$$a \in \text{sleep}_R(x) \implies xa \in \text{ignore}_R \quad (2)$$

Intuitively, a node xa is ignored if (1) any of its ancestors is ignored ($\text{ignore}_R(x)$), or (2) a is one of the ignored transitions at node x ($a \in \text{sleep}_R(x)$).

Finally, we obtain an actual reduction of a program P from a characterization of a reduction R by removing the ignored nodes from P , i.e. $P \setminus \text{ignore}_R$.

Lemma 6.5. *For all $R : \Sigma^* \rightarrow \text{Lin}(\Sigma)$, if P is a \sim_D -closed program then $P \setminus \text{ignore}_R$ is a D -reduction of P .*

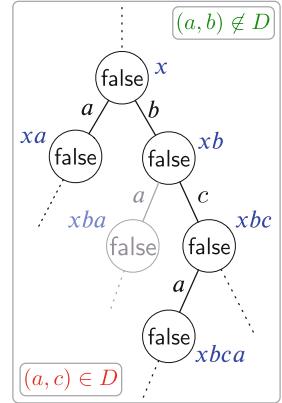


Fig. 3. Exploring from x with sleep sets.

The set of all such reductions is $\text{reduce}_D(P) = \{P \setminus \text{ignore}_R \mid R : \Sigma^* \rightarrow \mathcal{L}\text{in}(\Sigma)\}$.

Theorem 6.6. *For any regular language P , $\text{reduce}_D(P)$ is accepted by an LTA.*

Interestingly, every reduction in $\text{reduce}_D(P)$ is optimal in the sense that each reduction contains at most one representative of each equivalence class of \sim_D .

Theorem 6.7. *Fix some $P \subseteq \Sigma^*$ and $R : \Sigma^* \rightarrow \mathcal{L}\text{in}(\Sigma)$. For all $(x, y) \in P \setminus \text{ignore}_R$, if $x \sim_D y$ then $x = y$.*

7 Algorithms

Figure 4 illustrates the outline of our verification algorithm. It is a counterexample-guided abstraction refinement loop in the style of [12, 13, 17]. The key difference is that instead of checking whether some proof Π is a proof for the program P , it checks if there exists a reduction of the program P that Π proves correct.

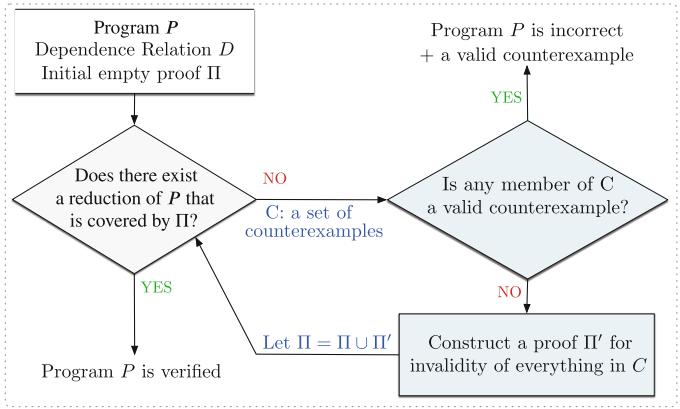


Fig. 4. Counterexample-guided refinement loop.

The algorithm relies on an oracle `INTERPOLATE` that, given a finite set of program traces C , returns a proof Π' , if one exists, such that $C \subseteq \mathcal{L}(\Pi')$. In our tool, we use Craig interpolation to implement the oracle `INTERPOLATE`. In general, since program traces are the simplest form of sequential programs (loop and branch free), any automated program prover that can handle proving them may be used.

The results presented in Sects. 5 and 6 give rise to the proof checking subroutine of the algorithm in Fig. 4 (i.e. the light grey test). Given a program DFA $A_P = (Q_P, \Sigma, \delta_P, q_{P0}, F_P)$ and a proof DFA $A_\Pi = (Q_\Pi, \Sigma, \delta_\Pi, q_{\Pi0}, F_\Pi)$ (obtained by determinizing Π_{NFA}), we can decide $\exists P' \in \text{reduce}_D(\mathcal{L}(A_P)). P' \subseteq \mathcal{L}(A_\Pi)$ by constructing an LTA $M_{P\Pi}$ for $\text{reduce}_D(\mathcal{L}(A_P)) \cap \mathcal{P}(\mathcal{L}(A_\Pi))$ and checking emptiness (Theorem 5.2).

7.1 Progress

The algorithm corresponding to Fig. 4 satisfies a weak progress theorem: none of the counterexamples from a round of the algorithm will ever appear in a

future counterexample set. This, however, is not strong enough to guarantee termination. Alternatively, one can think of the algorithm’s progress as follows. In each round new assertions are discovered through the oracle INTERPOLATE, and one can optimistically hope that one can finally converge on an existing target proof Π^* . The success of this algorithm depends on two factors: (1) the counterexamples used by the algorithm belong to $\mathcal{L}(\Pi^*)$ and (2) the proof that INTERPOLATE discovers for these counterexamples coincide with Π^* . The latter is a typical known wild card in software model checking, which cannot be guaranteed; there is plenty of empirical evidence, however, that procedures based on Craig Interpolation do well in approximating it. The former is a new problem for our refinement loop.

In a standard algorithm in the style of [12, 13, 17], the verification proof rule dictates that every program trace must be in $\mathcal{L}(\Pi^*)$. In our setting, we only require a subset (corresponding to some reduction) to be in $\mathcal{L}(\Pi^*)$. This means one cannot simply rely on program traces as *appropriate* counterexamples. Theorem 5.4 presents a solution to this problem. It ensures that we always feed INTERPOLATE some counterexample from Π^* and therefore guarantee progress.

Theorem 7.1 (Strong Progress). *Assume a proof Π^* exists for some reduction $P^* \in \mathcal{R}$ and INTERPOLATE always returns some subset of Π^* for traces in $\mathcal{L}(\Pi^*)$. Then the algorithm will terminate in at most $|\Pi^*|$ iterations.*

Theorem 7.1 ensures that the algorithm will never get into an infinite loop due to a bad choice of counterexamples. The condition on INTERPOLATE ensures that divergence does not occur due to the wrong choice of assertions by INTERPOLATE and without it any standard interpolation-based software model checking algorithm may diverge. The assumption that there exists a proof for a reduction of the program in the fixed set \mathcal{R} ensures that the proof checking procedure can verify the target proof Π^* once it is reached. Note that, in general, a proof may exist for a reduction of the program which is not in \mathcal{R} . Therefore, the algorithm is not complete with respect to all reductions, since checking the premises of SAFERED1 is undecidable as discussed in Sect. 4.

7.2 Faster Proof Checking Through Antichains

The state set of $M_{\mathcal{P}\Pi}$, the intersection of program and proof LTAs, has size $|Q_P \times \mathbb{B} \times \mathcal{P}(\Sigma) \times Q_\Pi|$, which is exponential in $|\Sigma|$. Therefore, even a linear emptiness test for this LTA can be computationally expensive. Antichains have been previously used [8] to optimize certain operations over NFAs that also suffer from exponential blowups, such as deciding universality and inclusion tests. The main idea is that these operations involve computing downwards-closed and upwards-closed sets according to an appropriate subsumption relation, which can be represented compactly as antichains. We employ similar techniques to propose a new emptiness check algorithm.

Antichains. The set of maximal elements of a set X with respect to some ordering relation \sqsubseteq is denoted $\max(X)$. The downwards-closure of a set X with

respect to \sqsubseteq is denoted $\lfloor X \rfloor$. An antichain is a set X where no element of X is related (by \sqsubseteq) to another. The maximal elements $\max(X)$ of a finite set X is an antichain. If X is downwards-closed then $\lfloor \max(X) \rfloor = X$.

The emptiness check algorithm for LTAs from [2] computes the set of *inactive* states (i.e. states which generate an empty language) and checks if the initial state is inactive. The set of inactive states of an LTA $M = (Q, \Delta, q_0)$ is defined as the smallest set $\text{inactive}(M)$ satisfying

$$\frac{\forall (q, B, \sigma) \in \Delta. \exists a. \sigma(a) \in \text{inactive}(M)}{q \in \text{inactive}(M)} \quad (\text{INACTIVE})$$

Alternatively, one can view $\text{inactive}(M)$ as the least fixed-point of a monotone (with respect to \sqsubseteq) function $F_M : \mathcal{P}(Q) \rightarrow \mathcal{P}(Q)$ where

$$F_M(X) = \{q \mid \forall (q, B, \sigma) \in \Delta. \exists a. \sigma(a) \in X\}.$$

Therefore, $\text{inactive}(M)$ can be computed using a standard fixpoint algorithm.

If $\text{inactive}(M)$ is downwards-closed with respect to some *subsumption relation* (\sqsubseteq) $\subseteq Q \times Q$, then we need not represent all of $\text{inactive}(M)$. The antichain $\max(\text{inactive}(M))$ of maximal elements of $\text{inactive}(M)$ (with respect to \sqsubseteq) would be sufficient to represent the entirety of $\text{inactive}(M)$, and can be exponentially smaller than $\text{inactive}(M)$, depending on the choice of relation \sqsubseteq .

A trivial way to compute $\max(\text{inactive}(M))$ is to first compute $\text{inactive}(M)$ and then find the maximal elements of the result, but this involves doing strictly more work than the baseline algorithm. However, observe that if F_M also preserves downwards-closedness with respect to \sqsubseteq , then

$$\begin{aligned} \max(\text{inactive}(M)) &= \max(\text{lfp}(F_M)) \\ &= \max(\text{lfp}(F_M \circ \lfloor - \rfloor \circ \max)) = \text{lfp}(\max \circ F_M \circ \lfloor - \rfloor) \end{aligned}$$

That is, $\max(\text{inactive}(M))$ is the least fixed-point of a function $F_M^{\max} : \mathcal{P}(Q) \rightarrow \mathcal{P}(Q)$ defined as $F_M^{\max}(X) = \max(F_M(\lfloor X \rfloor))$. We can calculate $\max(\text{inactive}(M))$ efficiently if we can calculate $F_M^{\max}(X)$ efficiently, which is true in the special case of the intersection automaton for the languages of our proof $\mathcal{P}(\mathcal{L}(\Pi))$ and our program $\text{reduce}_D(P)$, which we refer to as $M_{P\Pi}$.

We are most interested in the state space of $M_{P\Pi}$, which is $Q_{P\Pi} = (Q_P \times \mathbb{B} \times \mathcal{P}(\Sigma)) \times Q_\Pi$. Observe that states whose \mathbb{B} part is \top are always active:

Lemma 7.2. $((q_P, \top, S), q_\Pi) \notin \text{inactive}(M_{P\Pi})$ for all $q_P \in Q_P$, $q_\Pi \in Q_\Pi$, and $S \subseteq \Sigma$.

The state space can then be assumed to be $Q_{P\Pi} = (Q_P \times \{\perp\} \times \mathcal{P}(\Sigma)) \times Q_\Pi$ for the purposes of checking inactivity. The subsumption relation defined as the smallest relation $\sqsubseteq_{P\Pi}$ satisfying

$$S \subseteq S' \implies ((q_P, \perp, S), q_\Pi) \sqsubseteq_{P\Pi} ((q_P, \perp, S'), q_\Pi)$$

for all $q_P \in Q_P$, $q_\Pi \in Q_\Pi$, and $S, S' \subseteq \Sigma$, is a suitable one since:

Lemma 7.3. $F_{M_{P\pi}}$ preserves downwards-closedness with respect to $\sqsubseteq_{P\pi}$.

The function $F_{M_{P\pi}}^{\max}$ is a function over relations

$$F_{M_{P\pi}}^{\max} : \mathcal{P}((Q_P \times \{\perp\} \times \mathcal{P}(\Sigma)) \times Q_\pi) \rightarrow \mathcal{P}((Q_P \times \{\perp\} \times \mathcal{P}(\Sigma)) \times Q_\pi)$$

but in our case it is more convenient to view it as a function over functions

$$F_{M_{P\pi}}^{\max} : (Q_P \times \{\perp\} \times Q_\pi \rightarrow \mathcal{P}(\mathcal{P}(\Sigma))) \rightarrow (Q_P \times \{\perp\} \times Q_\pi \rightarrow \mathcal{P}(\mathcal{P}(\Sigma)))$$

Through some algebraic manipulation and some simple observations, we can define $F_{M_{P\pi}}^{\max}$ functionally as follows.

Lemma 7.4. For all $q_P \in Q_P$, $q_\pi \in Q_\pi$, and $X : Q_P \times \{\perp\} \times Q_\pi \rightarrow \mathcal{P}(\mathcal{P}(\Sigma))$,

$$F_{M_{P\pi}}^{\max}(X)(q_P, \perp, q_\pi) = \begin{cases} \{\Sigma\} & \text{if } q_P \in F_P \wedge q_\pi \notin F_\pi \\ \prod_{R \in \mathcal{L}in(\Sigma)} \bigsqcup_{\substack{a \in \Sigma \\ S \in X(q'_P, \perp, q'_\pi)}} S' & \text{otherwise} \end{cases}$$

where

$$\begin{aligned} q'_P &= \delta_P(q_P, a) & X \sqcap Y &= \max\{x \sqcap y \mid x \in X \wedge y \in Y\} \\ q'_\pi &= \delta_\pi(q_\pi, a) & X \sqcup Y &= \max(X \cup Y) \end{aligned}$$

$$S' = \begin{cases} \{(S \cup D(a)) \setminus \{a\}\} & \text{if } R(a) \setminus D(a) \subseteq S \\ \emptyset & \text{otherwise} \end{cases}$$

```

function Check( $A_P, A_\pi, D$ )
   $(Q_P, \Sigma, \delta_P, q_{0P}, F_P) \leftarrow A_P$ 
   $(Q_\pi, \Sigma, \delta_\pi, q_{0\pi}, F_\pi) \leftarrow A_\pi$ 
  function FMax( $X$ ) ( $(q_P, \perp, q_\pi)$ )
    if  $q_P \in F_P \wedge q_\pi \notin F_\pi$ 
    | return  $\{\Sigma\}$ 
     $X^\sqcap \leftarrow \{\Sigma\}$ 
    for  $R \in \mathcal{L}in(\Sigma)$ 
      |  $X^\sqcup \leftarrow \emptyset$ 
      | for  $a \in \Sigma, S \in X((\delta_P(q_P, a), \perp, \delta_\pi(q_\pi, a)))$ 
        | | if  $R(a) \setminus D(a) \subseteq S$ 
        | | |  $X^\sqcup \leftarrow X^\sqcup \sqcup \{(S \cup D(a)) \setminus \{a\}\}$ 
        | |  $X^\sqcap \leftarrow X^\sqcap \sqcap X^\sqcup$ 
    return  $X^\sqcap$ 
  return Fix(FMax) ( $(q_{0P}, \perp, q_{0\pi})$ )  $\neq \emptyset$ 

```

Algorithm 1. Proof checking algorithm

A full justification appears in [14]. Formulating $F_{M_{P\Pi}}^{\max}$ as a higher-order function allows us to calculate $\max(\text{inactive}(M_{P\Pi}))$ using efficient fixpoint algorithms like the one in [22]. Algorithm 1 outlines our proof checking routine. $\text{FIX} : ((A \rightarrow B) \rightarrow (A \rightarrow B)) \rightarrow (A \rightarrow B)$ is a procedure that computes the least fixpoint of its input. The algorithm simply computes the fixpoint of the function $F_{M_{P\Pi}}^{\max}$ as defined in Lemma 7.4, which is a compact representation of $\text{inactive}(M_{P\Pi})$ and checks if the start state of $M_{P\Pi}$ is in it.

Counterexamples. Theorem 5.4 states that a finite set of counterexamples exists whenever $\exists P' \in \text{reduce}_D(P). P' \subseteq \mathcal{L}(\Pi)$ does not hold. The proof of emptiness for an LTA, formed using rule INACTIVE above, is a finite tree. Each edge in the tree is labelled by an element of Σ (obtained from the existential in the rule) and the paths through this tree form the counterexample set. To compute this set, then, it suffices to remember enough information during the computation of $\text{inactive}(M)$ to reconstruct the proof tree. Every time a state q is determined to be inactive, we must also record the witness $a \in \Sigma$ for each transition $(q, B, \sigma) \in \Delta$ such that $\sigma(a) \in \text{inactive}(M)$.

In an antichain-based algorithm, once we determine a state q to be inactive, we simultaneously determine everything it subsumes (i.e. $\sqsubseteq q$) to be inactive as well. If we record unique witnesses for each and every state that q subsumes, then the space complexity of our antichain algorithm will be the same as the unoptimized version. The following lemma states that it is sufficient to record witnesses only for q and discard witnesses for states that q subsumes.

Lemma 7.5. *Fix some states q, q' such that $q' \sqsubseteq_{P\Pi} q$. A witness used to prove q is inactive can also be used to prove q' is inactive.*

Note that this means that the antichain algorithm soundly returns potentially fewer counterexamples than the original one.

7.3 Partition Optimization

The LTA construction for $\text{reduce}_D(P)$ involves a nondeterministic choice of linear order at each state. Since $|\mathcal{Lin}(\Sigma)|$ has size $|\Sigma|!$, each state in the automaton would have a large number of transitions. As an optimization, our algorithm selects ordering relations out of $\mathcal{Part}(\Sigma)$ (instead of $\mathcal{Lin}(\Sigma)$), defined as $\mathcal{Part}(\Sigma) = \{\Sigma_1 \times \Sigma_2 \mid \Sigma_1 \uplus \Sigma_2 = \Sigma\}$ where \uplus is disjoint union. This leads to a sound algorithm which is not complete with respect to sleep set reductions and trades the factorial complexity of computing $\mathcal{Lin}(\Sigma)$ for an exponential one.

8 Experimental Results

To evaluate our approach, we have implemented our algorithm in a tool called WEAVER written in Haskell. WEAVER accepts a program written in a simple imperative language as input, where the property is already encoded in the program in the form of *assume* statements, and attempts to prove the program

correct. The dependence relation for each input program is computed using a heuristic that ensures \sim_D -closedness. It is based on the fact that the shuffle product (i.e. parallel composition) of two \sim_D -closed languages is \sim_D -closed.

WEAVER employs two verification algorithms: (1) The total order algorithm presented in Algorithm 1, and (2) the variation with the partition optimization discussed in Sect. 7.3. It also implements multiple counterexample generation algorithms: (1) *Naive*: selects the first counterexample in the difference of the program and proof language. (2) *Progress-Ensuring*: selects a set of counterexamples satisfying Theorem 5.4. (3) *Bounded Progress-Ensuring*: selects a few counterexamples (in most cases just one) from the set computed by the progress-ensuring algorithm. Our experimentation demonstrated that in the vast majority of the cases, the bounded progress ensuring algorithm (an instance of the partition algorithm) is the fastest of all options. Therefore, all our reports in this section are using this instance of the algorithm.

For the larger benchmarks, we use a simple sound optimization to reduce the proof size. We declare the basic blocks of code as atomic, so that internal assertions need not be generated for them as part of the proof. This optimization is incomplete with respect to sleep set reductions.

Benchmarks. We use a set of sequential benchmarks from [24] and include additional sequential benchmarks that involve more interesting reductions in their proofs. We have a set of parallel benchmarks, which are beyond the scope of previous hypersafety verification techniques. We use these benchmarks to demonstrate that our technique/tool can seamlessly handle concurrency. These involve proving concurrency specific hypersafety properties such as determinism and equivalence of parallel and sequential implementations of algorithms. Finally, since the proof checking algorithm is the core contribution of this paper, we have a contrived set of instances to stress test our algorithm. These involve proving determinism of simple parallel-disjoint programs with various numbers of threads and statements per thread. These benchmarks have been designed to cause a combinatorial explosion for the proof checker and counterexample generation routines. More information on the benchmarks can be found in [14].

Evaluation

Due to space restrictions, it is not feasible to include a detailed account of all our experiments here, for over 50 benchmarks. A detailed table can be found in [14]. Table 1 includes a summary in the form of averages, and here, we discuss our top findings.

Proof construction time refers to the time spent to construct $\mathcal{L}(\Pi)$ from a given set of assertions Π and excludes the time to produce proofs for the counterexamples in a given round. **Proof checking time** is the time spent to check if the current proof candidate is strong enough for a reduction of the program. In the fastest instances (total time around 0.01 s), roughly equal time is spent in proof checking and proof construction. In the slowest instances, the total time is almost entirely spent in proof construction. In contrast, in our stress

Table 1. Experimental results averages for benchmark groups.

Benchmark group	Group count	Proof size	Number of refinement rounds	Proof construction time	Proof checking time	Total time
Looping programs of [24] 2-safety properties	5	63	12	46.69 s	0.1 s	47.03 s
Looping programs of [24] 3-safety properties	8	155	22	475.78 s	11.79 s	448.36 s
Loop-free programs of [24]	27	5	2	0.13 s	0.0004 s	0.15 s
Our sequential benchmarks	13	30	9	14.27 s	2.5 s	17.94 s
Our parallel benchmarks	7	31	8	17.95	0.56 s	18.63 s

tests (designed to stress the proof checking algorithm) the majority of the time is spent in proof checking. The time spent in proving counterexamples correct is negligible in all instances. **Proof sizes** vary from 4 assertions to 298 for the most complicated instance. Verification times are *correlated* with the final proof size; larger proofs tend to cause longer verification times.

Numbers of refinement rounds vary from 2 for the simplest to 33 for the most complicated instance. A small number of refinement rounds (e.g. 2) implies a fast verification time. But, for the higher number of rounds, a strong positive correlation between the number of rounds and verification time does not exist.

For our **parallel programs** benchmarks (other than our stress tests), the tool spends the majority of its time in proof construction. Therefore, we designed specific (unusual) parallel programs to stress test the proof checker. **Stress test** benchmarks are trivial tests of determinism of disjoint parallel programs, which can be proven correct easily by using the atomic block optimization. However, we force the tool to do the unnecessary hard work. These instances simulate the worst case theoretical complexity where the proof checking time and number of counterexamples grow exponentially with the number of threads and the sizes of the threads. In the largest instance, more than 99% of the total verification time is spent in proof checking. Averages are not very informative for these instances, and therefore are not included in Table 1.

Finally, WEAVER is only slow for verifying 3-safety properties of large looping benchmarks from [24]. Note that unlike the approach in [24], which starts from a default lockstep reduction (that is incidentally sufficient to prove these instances), we do not assume any reduction and consider them all. The extra time is therefore expected when the product programs become quite large.

9 Related Work

The notion of a k -safety hyperproperty was introduced in [7] without consideration for automatic program verification. The approach of reducing k -safety to 1-safety by self-composition is introduced in [5]. While theoretically complete, self-composition is not practical as discussed in Sect. 1. Product programs generalize the self-composition approach and have been used in verifying translation

validation [20], non-interference [16, 23], and program optimization [25]. A product of two programs P_1 and P_2 is semantically equivalent to $P_1 \cdot P_2$ (sequential composition), but is made easier to verify by allowing parts of each program to be interleaved. The product programs proposed in [3] allow lockstep interleaving exclusively, but only when the control structures of P_1 and P_2 match. This restriction is lifted in [4] to allow some non-lockstep interleavings. However, the given construction rules are non-deterministic, and the choice of product program is left to the user or a heuristic.

Relational program logics [6, 28] extend traditional program logics to allow reasoning about relational program properties, however automation is usually not addressed. Automatic construction of product programs is discussed in [10] with the goal of supporting procedure specifications and modular reasoning, but is also restricted to lockstep interleavings. Our approach does not support procedure calls but is fully automated and permits non-lockstep interleavings.

The key feature of our approach is the automation of the discovery of an appropriate program reduction and a proof combined. In this case, the only other method that compares is the one based on Cartesian Hoare Logic (CHL) proposed in [24] along with an algorithm for automatic verification based on CHL. Their proposed algorithm implicitly constructs a product program, using a heuristic that favours lockstep executions as much as possible, and then prioritizes certain rules of the logic over the rest. The heuristic nature of the search for the proof means that no characterization of the search space can be given, and no guarantees about whether an appropriate product program will be found. In contrast, we have a formal characterization of the set of explored product programs in this paper. Moreover, CHL was not designed to deal with concurrency.

Lipton [19] first proposed reduction as a way to simplify reasoning about concurrent programs. His ideas have been employed in a semi-automatic setting in [11]. Partial-order reduction (POR) is a class of techniques that reduces the state space of search by removing redundant paths. POR techniques are concerned with finding a single (preferably minimal) reduction of the input program. In contrast, we use the same underlying ideas to explore many program reductions simultaneously. The class of reductions described in Sect. 6 is based on the sleep set technique of Godefroid [15]. Other techniques exist [1, 15] that are used in conjunction with sleep sets to achieve minimality in a normal POR setting. In our setting, reductions generated by sleep sets are already optimal (Theorem 6.7). However, employing these additional POR techniques may propose ways of optimizing our proof checking algorithm by producing a smaller reduction LTA.

References

1. Abdulla, P.A., Aronis, S., Jonsson, B., Sagonas, K.: Source sets: a foundation for optimal dynamic partial order reduction. J. ACM (JACM) **64**(4), 25 (2017)

2. Baader, F., Tobies, S.: The inverse method implements the automata approach for modal satisfiability. In: Goré, R., Leitsch, A., Nipkow, T. (eds.) IJCAR 2001. LNCS, vol. 2083, pp. 92–106. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45744-5_8
3. Barthe, G., Crespo, J.M., Kunz, C.: Relational verification using product programs. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 200–214. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21437-0_17
4. Barthe, G., Crespo, J.M., Kunz, C.: Beyond 2-safety: asymmetric product programs for relational program verification. In: Artemov, S., Nerode, A. (eds.) LFCS 2013. LNCS, vol. 7734, pp. 29–43. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35722-0_3
5. Barthe, G., D’argenio, P.R., Rezk, T.: Secure information flow by self-composition. *Math. Struct. Comput. Sci.* **21**(6), 1207–1252 (2011)
6. Benton, N.: Simple relational correctness proofs for static analyses and program transformations. In: ACM SIGPLAN Notices, vol. 39, pp. 14–25. ACM (2004)
7. Clarkson, M.R., Schneider, F.B.: Hyperproperties. In: 21st IEEE Computer Security Foundations Symposium, pp. 51–65. IEEE (2008)
8. 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.) CAV 2006. LNCS, vol. 4144, pp. 17–30. Springer, Heidelberg (2006). https://doi.org/10.1007/11817963_5
9. Diekert, V., Métivier, Y.: Partial commutation and traces. In: Rozenberg, G., Salomaa, A. (eds.) Handbook of Formal Languages, pp. 457–533. Springer, Heidelberg (1997). https://doi.org/10.1007/978-3-642-59126-6_8
10. Eilers, M., Müller, P., Hitz, S.: Modular product programs. In: Ahmed, A. (ed.) ESOP 2018. LNCS, vol. 10801, pp. 502–529. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89884-1_18
11. Elmas, T., Qadeer, S., Tasiran, S.: A calculus of atomic actions. In: ACM SIGPLAN Notices, vol. 44, pp. 2–15. ACM (2009)
12. Farzan, A., Kincaid, Z., Podelski, A.: Inductive data flow graphs. In: ACM SIGPLAN Notices, vol. 48, pp. 129–142. ACM (2013)
13. Farzan, A., Kincaid, Z., Podelski, A.: Proof spaces for unbounded parallelism. In: ACM SIGPLAN Notices, vol. 50, pp. 407–420. ACM (2015)
14. Farzan, A., Vandikas, A.: Reductions for automated hypersafety verification (2019)
15. Godefroid, P. (ed.): Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Expllosion Problem, vol. 1032. Springer, Heidelberg (1996). <https://doi.org/10.1007/3-540-60761-7>
16. Goguen, J.A., Meseguer, J.: Security policies and security models. In: 1982 IEEE Symposium on Security and Privacy, p. 11. IEEE (1982)
17. Heizmann, M., Hoenicke, J., Podelski, A.: Refinement of trace abstraction. In: Palsberg, J., Su, Z. (eds.) SAS 2009. LNCS, vol. 5673, pp. 69–85. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03237-0_7
18. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation, 3rd edn. Addison-Wesley Longman Publishing Co. Inc., Boston (2006)
19. Lipton, R.J.: Reduction: a method of proving properties of parallel programs. *Commun. ACM* **18**(12), 717–721 (1975)
20. Pnueli, A., Siegel, M., Singerman, E.: Translation validation. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, pp. 151–166. Springer, Heidelberg (1998). <https://doi.org/10.1007/BFb0054170>

21. Popeea, C., Rybalchenko, A., Wilhelm, A.: Reduction for compositional verification of multi-threaded programs. In: Formal Methods in Computer-Aided Design (FMCAD), 2014, pp. 187–194. IEEE (2014)
22. Pottier, F.: Lazy least fixed points in ML (2009)
23. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE J. Sel. Areas Commun. **21**(1), 5–19 (2003)
24. Sousa, M., Dillig, I.: Cartesian hoare logic for verifying k-safety properties. In: ACM SIGPLAN Notices, vol. 51, pp. 57–69. ACM (2016)
25. Sousa, M., Dillig, I., Vytiniotis, D., Dillig, T., Gkantsidis, C.: Consolidation of queries with user-defined functions. In: ACM SIGPLAN Notices, vol. 49, pp. 554–564. ACM (2014)
26. Terauchi, T., Aiken, A.: Secure information flow as a safety problem. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 352–367. Springer, Heidelberg (2005). https://doi.org/10.1007/11547662_24
27. Vardi, M.Y., Wolper, P.: Reasoning about infinite computations. Inf. Comput. **115**(1), 1–37 (1994)
28. Yang, H.: Relational separation logic. Theor. Comput. Sci. **375**(1–3), 308–334 (2007)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Automated Synthesis of Secure Platform Mappings

Eunsuk Kang¹(✉), Stéphane Lafortune²,
and Stavros Tripakis³



¹ Carnegie Mellon University, Pittsburgh, USA
eskang@cmu.edu

² University of Michigan, Ann Arbor, USA
stephane@umich.edu

³ Northeastern University, Boston, USA
stavros@northeastern.edu

Abstract. System development often involves decisions about how a high-level design is to be implemented using primitives from a low-level platform. Certain decisions, however, may introduce undesirable behavior into the resulting implementation, possibly leading to a violation of a desired property that has already been established at the design level. In this paper, we introduce the problem of *synthesizing a property-preserving platform mapping*: synthesize a set of implementation decisions ensuring that a desired property is preserved from a high-level design into a low-level platform implementation. We formalize this synthesis problem and propose a technique for generating a mapping based on symbolic constraint search. We describe our prototype implementation, and two real-world case studies demonstrating the applicability of our technique to the synthesis of secure mappings for the popular web authorization protocols OAuth 1.0 and 2.0.

1 Introduction

When building a complex software system, one may begin by coming up with an abstract design, and then construct an implementation that conforms to this design. In practice, there are rarely enough time and resources available to build an implementation from scratch, and so this process often involves reuse of an existing *platform*—a collection of generic components, data structures, and libraries that are used to build an application in a particular domain.

The benefits of reuse also come with potential risks. A typical platform exhibits its own complex behavior, including subtle interactions with the environment that may be difficult to anticipate and reason about. Typically, the developer must work with the platform as it exists, and is rarely given the luxury of being able to modify it and remove unwanted features. For example, when building a web application, a developer must work with a standard browser and take into account all its features and security vulnerabilities. As a result, achieving an implementation that perfectly conforms to the design—in the traditional notion of behavioral refinement [20]—may be too difficult

This work has been supported by NSF award CNS-1801546.

© The Author(s) 2019
I. Dillig and S. Tasiran (Eds.): CAV 2019, LNCS 11561, pp. 219–237, 2019.
https://doi.org/10.1007/978-3-030-25540-4_12

in practice. Worse, the resulting implementation may not necessarily preserve desirable properties that have already been established at the level of design.

These risks are especially evident in applications where security is a major concern. For example, OAuth 2.0, a popular authorization protocol subjected to rigorous and formal analysis at an abstract level [9,33,42], has been shown to be vulnerable to attacks when implemented on a web browser or a mobile device [10,39,41]. Many of these vulnerabilities are not due to simple programming errors: They arise from logical flaws that involve a subtle interaction between the protocol logic and the details of the underlying platform. Unfortunately, OAuth itself does not explicitly guard against these flaws, since it is intended to be a *generic, abstract* protocol that deliberately omits details about potential platforms. On the other hand, anticipating and mitigating against these risks require an in-depth understanding of the platform and security expertise, which many developers do not possess.

This paper proposes an approach to help developers overcome these risks and achieve an implementation that preserves desired properties. In particular, we formulate this task as the problem of automatically synthesizing a *property-preserving platform mapping*: A set of implementation decisions ensuring that a desired property is preserved from a high-level design into a low-level platform implementation.

Our approach builds on the prior work of Kang et al. [28], which proposes a modeling and verification framework for reasoning about security attacks across multiple levels of abstraction. The central notion in this framework is that of a *mapping*, which captures a developer’s decisions about how abstract system entities are to be realized in terms of their concrete counterparts. In this paper, we fix a bug in the formalization of mapping in [28] and extend the framework of [28] with the novel problem of synthesizing a property-preserving mapping. In addition, we present an algorithmic technique for performing this synthesis task. Our technique, inspired by the highly successful paradigms of *sketching* and *syntax-guided synthesis* [3,26,37,38], takes a *constraint generalization* approach to (1) quickly prune the search space and (2) produce a solution that is *maximal* (i.e., a largest set of mappings that preserve a given property).

We have built a prototype implementation of the synthesis technique. Our tool accepts a high-level design model, a desired system property (both specified by the developer), and a model of a low-level platform (built and maintained separately by a domain expert). The tool then produces a maximal set of mappings (if one exists) that would ensure that the resulting platform implementation preserves the given property. We have successfully applied our tool to synthesize property-preserving mappings for two non-trivial case studies: the authentication protocols OAuth 1.0 and 2.0 implemented on top of HTTP. Our results are promising: The implementation decisions captured by our synthesized mappings describe effective mitigations against some of the common vulnerabilities that have been found in deployed OAuth implementations [39,41].

The contributions of this paper include: a formal treatment of mapping, including a correction in the original definition [28] (Sect. 2); a formulation of the *mapping synthesis problem*, a novel approach for ensuring the preservation of a property between a high-level design and its platform implementation (Sect. 3); a technique for automatically synthesizing mappings based on symbolic constraint search (Sect. 4); and a prototype implementation of the synthesis technique along with a real-world case study

demonstrating the feasibility of this approach (Sect. 5). We conclude with a discussion of related work (Sect. 6).

2 Mapping Composition

Our approach builds on the modeling and verification framework proposed by Kang et al. [28], which is designed to allow modular reasoning about behavior of processes across multiple abstraction layers. In this framework, a trace-based semantic model (based on CSP [21]) is extended to represent events as *sets of labels*, and includes a new composition operator based on the notion of *mappings*, which relate event labels from one abstraction layer to another. In this section, we present the essential elements of this framework.

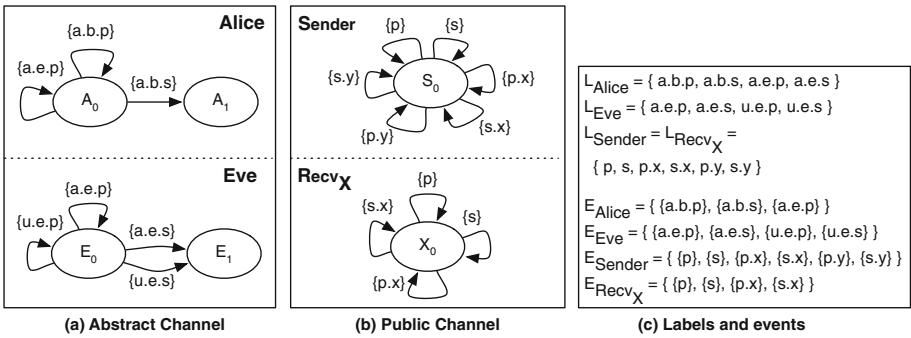


Fig. 1. A pair of high-level (abstract) and low-level (public) communication models. Note that each event is a *set of labels*, where each label describes one possible representation of the event.

Running Example. Consider a simple example involving communication of messages among a set of processes. In our modeling approach, the communication of a message is represented by labels of the form *sender.receiver.message*. For example, label *a.e.p* represents Alice sending Eve a public, non-secret message. Similarly, *a.b.s* represents Alice sending a secret message to another process (b for Bob, for example). In this system, Alice is unwilling to share its secret with Eve; in Fig. 1(a), this is modeled by the absence of any transition on event *{a.e.s}* in the Alice process.

Eve is a malicious character whose goal is to learn Alice's secret. Beside *a.e.p* and *a.e.s*, Eve is associated with two additional labels, *u.e.p* and *u.e.s*, which represent receiving a public or secret message, respectively, through some *unknown* sender *u*. Conceptually, these two latter labels can be regarded as *side channels* [30] that Eve uses to obtain information.

A desirable property of this abstract communication system is that Eve should never be able to learn Alice's secret¹. In this case, it can be easily observed that the property holds, since Alice, by design, never sends the secret to Eve.

¹ A formalization of this property is provided later in this section.

The model in Fig. 1(b) describes communication over a low-level public channel that is shared among all processes. A message sent over this channel may be encrypted using a key, as captured by labels of the form *message.key*. For instance, *p.x* and *s.x* represent the transmission of a public and secret message, respectively, using key *x*. A message may also be sent in plaintext by omitting an encryption key (e.g., label *s* represents the plaintext transmission of a secret). Each receiver on the public channel is assumed to have knowledge of only a single key; for instance, Recv_X only knows key *y* and thus cannot receive messages that are encrypted using key *y* (i.e., labels *p.y* and *s.y* do not appear in events of Recv_X).

Suppose that we wish to reason about the behavior of the abstract communication system from Fig. 1(a) when it is implemented over the public channel in Fig. 1(b). In particular, in the low-level implementation, Eve and other processes (e.g., Bob) are required to share the same channel, no longer benefitting from the separation provided by the abstraction in Fig. 1(a). Does the property of the abstract communication hold in every possible implementation? If not, which decisions ensure that Alice’s secret remains protected from Eve? We formulate these questions as the problem of synthesizing a *property-preserving mapping* between a pair of high-level and low-level models.

Events, Traces, and Processes. Let L be a potentially infinite set of labels. An *event* e is a finite, non-empty set of labels: $e \in E(L)$, where $E(L)$ is the set of all finite subsets of L except the empty set \emptyset . Let S^* be the set of all finite sequences of elements of set S . A *trace* t is a finite sequence of events: $t \in T(L)$, where $T(L)$ is the set of all traces over L (i.e., $T(L) = (E(L))^*$). The empty trace is denoted by $\langle \rangle$, and the trace consisting of a sequence of events e_1, e_2, \dots is denoted $\langle e_1, e_2, \dots \rangle$. If t and t' are traces, then $t \cdot t'$ is the trace obtained by concatenating t and t' . Note that $\langle \rangle \cdot t = t \cdot \langle \rangle = t$ for any trace t .

Let t be a trace over set of labels L , and let $A \subseteq L$ be a subset of L . The *projection* of t onto A , denoted $t \upharpoonright A$, is defined as follows:

$$\langle \rangle \upharpoonright A = \langle \rangle \quad (\langle e \rangle \cdot t) \upharpoonright A = \begin{cases} \langle e \cap A \rangle \cdot (t \upharpoonright A) & \text{if } e \cap A \neq \emptyset \\ (t \upharpoonright A) & \text{otherwise} \end{cases}$$

For example, if $t = \langle \{a\}, \{a, c\}, \{b\} \rangle$, then $t \upharpoonright \{a, b\} = \langle \{a\}, \{a\}, \{b\} \rangle$ and $t \upharpoonright \{b, c\} = \langle \{c\}, \{b\} \rangle$.

A *process* P is defined as a triple (L_P, E_P, T_P) . The *labels* of process P , $L_P \subseteq L$, is the set of all labels appearing in P , and $E_P \subseteq E(L)$ is the set of events that *may* appear in traces of P , which are denoted by $T_P \subseteq T(L)$. We assume traces in every process P to be *prefix-closed*; i.e., $\langle \rangle \in T_P$ and for every non-empty trace $t' = t \cdot \langle e \rangle \in T_P$, $t \in T_P$.

Parallel Composition. A pair of processes P and Q synchronize with each other by performing events e_1 and e_2 , respectively, if these two events share at least one label. In their parallel composition, denoted $P \parallel Q$, this synchronization is represented by a new event e' that is constructed as the union of e_1 and e_2 (i.e., $e' = e_1 \cup e_2$).

Formally, let $P = (L_P, E_P, T_P)$ and $Q = (L_Q, E_Q, T_Q)$ be a pair of processes. Their parallel composition is defined as follows:

$$\begin{aligned} E_{P \parallel Q} &= \{e \in E(L_P \cup L_Q) \mid \text{eventCond}(e, P) \wedge \text{eventCond}(e, Q) \wedge \text{syncCond}(e)\} \\ T_{P \parallel Q} &= \{t \in (E_{P \parallel Q})^* \mid (t \upharpoonright L_P) \in T_P \wedge (t \upharpoonright L_Q) \in T_Q\} \end{aligned} \tag{Def. 1}$$

where $L_{P||Q} = L_P \cup L_Q$, predicate *eventCond* is defined as

$$\text{eventCond}(e, P) \equiv e \cap L_P = \emptyset \vee e \cap L_P \in E_P$$

and a condition on synchronization, *syncCond*, is defined as

$$\text{syncCond}(e) \equiv e \subseteq L_P - L_Q \vee e \subseteq L_Q - L_P \vee (\exists a \in e : a \in L_P \cap L_Q) \quad (\textbf{Cond. 1})$$

The definition of $T_{P||Q}$ states that if we take a trace t in the composite process and ignore labels that appear only in Q , then the resulting trace must be a valid trace of P (and symmetrically for Q). The condition (**Cond. 1**) is imposed on every event appearing in $T_{P||Q}$ to ensure that an event performed together by P and Q contains at least one common label shared by both processes.

This type of parallel composition can be seen as a generalization of the parallel composition of CSP [21], from single labels to *sets* of labels. That is, the CSP parallel composition is the special case of the composition of **Def. 1** where every event is a singleton (i.e., it contains exactly one label). Note that if event e contains exactly one label a , then a must belong to the alphabet of P or that of Q , which means $\text{syncCond}(e)$ always evaluates to true. The resulting expression in that case

$$T_{P||Q} = \{t \in T(L_P \cup L_Q) \mid (t \upharpoonright L_P) \in T_P \wedge (t \upharpoonright L_Q) \in T_Q\}$$

is equivalent to the definition of parallel composition in CSP [21, Sec. 2.3.3].

Mapping Composition. A *mapping* m over set of labels L is a partial function $m : L \rightarrow L$. Informally, $m(a) = b$ stipulates that every event that contains a as a label is to be assigned b as an additional label. We sometimes use the notations $a \mapsto_m b$ or $(a, b) \in m$ as alternatives to $m(a) = b$. When we write $m(a) = b$ we mean that $m(a)$ is defined and is equal to b . The *empty* mapping, denoted $m = \emptyset$, is the partial function $m : L \rightarrow L$ which is undefined for all $a \in L$.

Mapping composition allows a pair of processes to interact with each other over distinct labels. Formally, consider two processes $P = (L_P, E_P, T_P)$ and $Q = (L_Q, E_Q, T_Q)$, and let $L = L_P \cup L_Q$. Given mapping $m : L \rightarrow L$, the *mapping composition* $P \parallel_m Q$ is defined as follows:

$$\begin{aligned} E_{P \parallel_m Q} &= \{e \in E(L_P \cup L_Q) \mid \text{eventCond}(e, P) \wedge \text{eventCond}(e, Q) \wedge \\ &\quad \text{syncCond}'(e) \wedge \text{mapCond}(e, m)\} \\ T_{P \parallel_m Q} &= \{t \in (E_{P \parallel_m Q})^* \mid (t \upharpoonright L_P) \in T_P \wedge (t \upharpoonright L_Q) \in T_Q\} \end{aligned} \quad (\textbf{Def. 2})$$

where $L_{P \parallel_m Q} = L_P \cup L_Q$, and *syncCond'*(e) and *mapCond*(e, m) are defined as:

$$\begin{aligned} \text{syncCond}'(e) &\equiv \text{syncCond}(e) \vee (\exists a \in e \cap L_P, \exists b \in e \cap L_Q : m(a) = b \vee m(b) = a) \\ \text{mapCond}(e, m) &\equiv (\forall a \in e : a \in \text{dom}(m) \Rightarrow m(a) \in e) \end{aligned}$$

where *dom*(m) is the domain of function m . Compared to **Def. 1**, the additional disjunct in *syncCond'*(e) allows P and Q to synchronize even when they do not share any label,

if at least one pair of their labels are mapped to each other in m . The predicate $mapCond$ ensures that if an event e contains a label a and m is defined over a , then e also contains the label that a is mapped to.

Note that **Def. 2** is different from the definition of mapping composition in [28], and corrects a flaw in the latter. In particular, the definition in [28] omits condition $syncCond'$, which permits the undesirable case in which events e_1 and e_2 from P and Q are synchronized into union $e = e_1 \cup e_2$ even when the events do not share any label.

Example. Let P and Q be the abstract and public channel communication models from Fig. 1(a) and (b), respectively. The property that Eve never learns Alice's secret can be stated as follows:

$$\Phi \equiv \neg(\exists e \in E(L) : l_1, l_2 \in e : l_1 = a.*.s \wedge l_2 = *.e.s)$$

where $*$ $\in \{a, b, e, u\}$. In other words, Eve should never be able to engage in an event that involves the transmission of Alice's secret. From Fig. 1(a), it can be observed that $P = \text{Alice} \parallel \text{Eve} \models \Phi$.

Suppose that we decide on a simple implementation scheme where the abstract messages sent by Alice are transmitted over the public channel in plaintext; this decision can be encoded as a mapping, m_1 , where each abstract label (i.e., L_{Alice} in Fig. 1(c)) is mapped to concrete label p or s as follows:

$$a.b.p, a.e.p, u.e.p \mapsto_{m_1} p \quad a.b.s, a.e.s, u.e.s \mapsto_{m_1} s$$

The resulting implementation can be constructed as process $I_{m_1} \equiv (\text{Alice} \parallel_{m_1} \text{Sender}) \parallel (\text{Eve} \parallel_{m_1} \text{Recv}_X)$. Due to the definition of mapping composition (**Def. 2**), the following event may appear in a trace of the overall composite process:

$$\langle \{a.b.s, s, a.e.s\} \rangle \in T_{I_{m_1}}$$

Note that this trace is a violation of the above property (i.e., $I_{m_1} \not\models \Phi$). This can be seen as an example of *abstraction violation*: As a result of decisions in m_1 , a.b.s and u.e.s now share the same underlying representation (s), and Eve is able to engage in an event with a label (a.b.s) that was not previously available to it in the abstract model.

Properties of the Mapping Composition Operator. Mapping composition is a generalization of parallel composition: The latter is a special case of mapping composition where the given mapping is empty:

Lemma 1. *Given a pair of processes P and Q , if $m = \emptyset$ then $P \parallel_m Q = P \parallel Q$.*

Commutativity. The proposed mapping composition operator is commutative: i.e., $P \parallel_m Q = Q \parallel_m P$. This property can be inferred from the fact that **Def. 2** is symmetric with respect to P and Q . It follows that by being a special case of mapping composition, the parallel composition operator is also commutative.

Associativity. The mapping composition operator is associative under the following conditions on the alphabets of involved processes and mappings:

Theorem 1. *Given processes P , Q , and R , let $X = (P \parallel_{m_1} Q) \parallel_{m_2} R$ and $Y = P \parallel_{m_3} (Q \parallel_{m_4} R)$. If $E_X = E_Y$, then $X = Y$.*

Proof. Available in the extended version of this paper [27].

3 Synthesis Problems

The *mapping verification problem* is to check, given processes P and Q , mapping m , and specification Φ , whether $(P\|_m Q) \models \Phi$. This problem was studied by Kang et al. [28]. In this paper, we introduce and study, for the first time to our knowledge, the problem of *mapping synthesis*. We begin with a simple formulation of the problem and then generalize it. We will not define what exactly the specification Φ may be, neither the satisfaction relation \models , as the mapping synthesis problems defined below are generic and can work with any type of specification or satisfaction relation. In Sect. 5.1, we discuss how this generic framework is instantiated in our implementation.

Problem 1 (Mapping Synthesis). Given processes P and Q , and specification Φ , find, if it exists, a mapping m such that $(P\|_m Q) \models \Phi$. We call such an m a *valid* mapping.

Note that if Φ is a *trace* property [2, 29], this problem can be stated as a $\exists \forall$ problem; that is, finding a witness m to the formula $\exists m : \forall t \in T_{P\|_m Q} : t \in \Phi$.

Instead of synthesizing m from scratch, the developer may wish to express their partial system knowledge as a given *constraint*, and ask the synthesis tool to generate a mapping that adheres to this constraint. For instance, given labels $a, b, c \in L$, one may express a constraint that a must be mapped to either b or c as part of every valid mapping; this gives rise to two possible candidate mappings, m_1 and m_2 , where $m_1(a) = b$ and $m_2(a) = c$. Formally, let M be the set of all possible mappings between labels L . A *mapping constraint* $C \subseteq M$ is a set of mappings that are considered legal candidates for a final, synthesized valid mapping. Then, the problem of synthesizing a mapping given a constraint can be formulated as follows:

Problem 2 (Generalized Mapping Synthesis). Given processes P and Q , specification Φ , and mapping constraint C , find, if it exists, a valid mapping m such that $m \in C$.

Note that Problem 1 is a special case of Problem 2 where $C = M$. The synthesis problem can be further generalized to one that involves synthesizing a constraint that contains a *set* of valid mappings:

Problem 3 (Mapping Constraint Synthesis). Given processes P and Q , specification Φ , and mapping constraint C , generate, if it exists, a non-empty set of valid mappings C' such that $C' \subseteq C$. We call such a C' valid with respect to P, Q, Φ and C .

A procedure for solving Problem 3 can be used to solve Problem 2: Having generated constraint C' , we can pick any mapping $m \in C'$. Such an m is guaranteed to be valid and also to belong in C .

In practice, it is desirable for C' to be as large as possible while still being valid, as it provides more implementation choices (i.e., possible mappings). In particular, we say that a mapping constraint C' is *maximal* with respect to P, Q, Φ , and C if and only if (1) C' is valid with respect to P, Q, Φ , and C , and (2) there exists no other constraint C'' such that C'' is also valid w.r.t. P, Q, Φ, C , and $C' \subseteq C''$. Then, our final synthesis problem can be stated as follows:

Problem 4 (Maximal Constraint Synthesis). Given processes P and Q , property Φ , and constraint C , generate, if it exists, a maximal constraint C' with respect to P, Q, Φ, C .

If found, C' is a *local* optimal solution. In general, there may be multiple maximal constraints for given P , Q , Φ , and C .

Example. Back to our running example, an alternative implementation of the abstract communication model over the public channel involves encrypting messages sent by Alice to Bob using a key (y) that Eve does not possess; this decision can be encoded as the following *valid* mapping m_2 :

$$\text{a.b.p} \mapsto_{m_2} \text{p.y} \quad \text{a.b.s} \mapsto_{m_2} \text{s.y} \quad \text{a.e.p} \mapsto_{m_2} \text{p.x} \quad \text{a.e.s} \mapsto_{m_2} \text{s.y}$$

Since Eve cannot read messages encrypted using key y , she is unable to obtain Alice's secret over the public channel; thus, $I_{m_2} \models \Phi$, where $I_{m_2} \equiv (\text{Alice} \parallel_{m_2} \text{Sender}) \parallel (\text{Eve} \parallel_{m_2} \text{Recv}_X)$.

The following mapping, m_3 , which leaves non-secret messages unencrypted in the low-level channel (as p), is also valid with respect to Φ :

$$\text{a.b.p} \mapsto_{m_2} \text{p} \quad \text{a.b.s} \mapsto_{m_2} \text{s.y} \quad \text{a.e.p} \mapsto_{m_2} \text{p} \quad \text{a.e.s} \mapsto_{m_2} \text{s.y}$$

since Eve being able to read non-secret messages does not violate the property. Thus, the developer may choose either m_2 or m_3 to implement the abstract channel and ensure that Alice's secret remains protected from Eve. In other words, $C_1 = \{m_2, m_3\}$ is a valid (but not necessarily maximal) mapping constraint with respect to the desired property. Furthermore, C_1 is arguably more desirable than another constraint $C_2 = \{m_2\}$, since the former gives the developer more implementation choices than the latter does.

4 Synthesis Technique

Mapping Representation. In our approach, mappings are represented *symbolically* as logical expressions over variables that correspond to labels being mapped. The symbolic representation has the following advantages over an explicit one (where the entries of mapping m are enumerated explicitly): (1) it provides a succinct representation of implementation decisions to the developer (which is especially important as the size of the mapping grows large) and (2) it allows the user to specify partial implementation decisions (i.e., given constraint C) in a *declarative* manner.

We adopt the symbolic representation and, inspired by SyGuS [3], use a *syntactic* approach where the space of candidate mapping constraints is restricted to expressions that can be constructed from a given grammar. Our grammar is specified as follows:

$$\begin{aligned} \text{Term} &:= \text{Var} \mid \text{Const} & \text{Assign} &:= (\text{Term} = \text{Term}) \\ \text{Expr} &:= \text{Assign} \mid \neg \text{Assign} \mid \text{Assign} \Rightarrow \text{Assign} \mid \text{Expr} \wedge \text{Expr} \end{aligned}$$

where Var is a set of variables that represent parameters inside a label, and Const is the set of constant values. Intuitively, this grammar captures implementation decisions that involve assignments of parameters in an abstract label to their counterparts in a concrete label (represented by the equality operator “ $=$ ”). A logical implication is used to construct a conditional assignment of a parameter.

A mapping constraint is symbolically represented as a set of predicates, each of the form $\mathcal{X}(abs, conc)$ over *symbolic* labels *abs* and *conc*, where *abs* represents the label being mapped to *conc*. The body of each predicate is constructed as an expression from the above grammar. For example, let *abs* = *a.b.msg* be a symbolic encoding of labels that represent Alice communicating to Eve, with variable *msg* corresponding to the message being sent; similarly, let *conc* = *msg'.key* be a symbolic label in the public channel model, where *msg'* and *key* correspond to the message being transmitted and the key used to encrypt it (if any). Then, the expression

$$\mathcal{X}(a.b.msg, msg'.key) \equiv msg = msg' \wedge (msg = s \Rightarrow key = y)$$

states that (1) parameter *msg* in the abstract label must be equal to that in the concrete label (i.e., the message being transmitted must be preserved during the mapping) and (2) if the message is a secret, key *y* must be used to encrypt it in the implementation.

The set of mappings that predicate $\mathcal{X}(abs, conc)$ represents is defined as:

$$C = \{m : L \rightarrow L \mid \forall abs \in L : (abs \in \text{dom}(m) \Leftrightarrow \exists conc \in L : \mathcal{X}(abs, conc)) \wedge (abs \in \text{dom}(m) \Rightarrow \mathcal{X}(abs, m(abs)))\}$$

That is, a mapping *m* is allowed by $\mathcal{X}(abs, conc)$ if and only if for each label *abs*, (1) *m* is defined over *abs* if and only if there exists some label *conc* for which $\mathcal{X}(abs, conc)$ evaluates to true, and (2) *m* maps *abs* to such a label *conc*.

Algorithmic Considerations. To ensure that the algorithm terminates, the set of expressions that may be constructed using the given grammar is restricted to a finite set, by bounding the domains of data types (e.g., distinct messages and keys in our running example) and the size of expressions. We also assume the existence of a verifier that is capable of checking whether a candidate mapping satisfies a given specification Φ . The verifier implements function $\text{verify}(C, P, Q, \Phi)$ which returns *OK* if and only if every mapping allowed by constraint *C* is valid with respect to *P, Q, Φ*.

Generalization Algorithm. Once we limit the number of candidate expressions to be finite, we can use a brute-force algorithm to enumerate and check those candidates one by one. However, this naive algorithm is likely to suffer from scalability issues. Thus, we present an algorithm that takes a generalization-based approach to identify and prune undesirable parts of the search space. A key insight is that only a few implementation decisions—captured by some *minimal subset* of the entries in a mapping—may be sufficient to imply that the resulting implementation will be invalid. Thus, given some invalid mapping, the algorithm attempts to identify this minimal subset and construct a larger constraint *C_{bad}* that is guaranteed to contain only invalid mappings.

The outline of the algorithm is shown in Fig. 2. The function *synthesize* takes four inputs: processes *P* and *Q*, specification Φ , and a user-specified mapping constraint *C*. It also maintains a set of constraints *X*, which keeps track of “bad” regions of the search space that do not contain any valid mappings.

In each iteration, the algorithm selects some mapping *m* from *C* (line 3) and checks whether it belongs to one of the constraints in *X* (meaning, the mapping is guaranteed to result in an invalid implementation). If so, it is simply discarded (lines 4–5).

Otherwise, the verifier is used to check whether m is valid with respect to Φ (line 7). If so, then *generalize* is invoked to produce a maximal mapping constraint $C_{maximal}$, which represents the largest set that contains $\{m\}$, is contained in C , and is valid with respect to P, Q, Φ (line 9). If, on the other hand, m is invalid (i.e., it fails to preserve Φ), then *generalize* is invoked to compute the largest superset C_{bad} of $\{m\}$ that contains only invalid mappings (i.e., those that satisfy $\neg\Phi$). The set C_{bad} is then added to X and used to prune out subsequent, invalid candidates (line 13).

```

1  fun synthesize( $P, Q, \Phi, C$ )
2     $X = \{\}$ 
3    for  $m \in C$  do
4      if  $\exists C_{bad} \in X : m \in C_{bad}$  then
5        skip
6      end
7       $result \leftarrow verify(\{m\}, P, Q, \Phi)$ 
8      if  $result = OK$  then
9         $C_{maximal} \leftarrow generalize(\{m\}, P, Q, \Phi, C)$ 
10       return  $C_{maximal}$ 
11     else
12        $C_{bad} \leftarrow generalize(\{m\}, P, Q, \neg\Phi, C)$ 
13        $X \leftarrow X \cup \{C_{bad}\}$ 
14     end
15   end
16   return none
17 end

18 fun generalize( $C', P, Q, \Phi, C$ )
19    $K \leftarrow decompose(C')$ 
20   for  $k \in K$  do
21      $C_{relaxed} \leftarrow relax(C', k)$ 
22      $result \leftarrow verify(C_{relaxed}, P, Q, \Phi)$ 
23     if  $result = OK \wedge C_{relaxed} \subseteq C$  then
24        $C' \leftarrow C_{relaxed}$ 
25     end
26   end
27   return  $C'$ 
28 end
```

Fig. 2. An algorithm for synthesizing a maximal mapping constraint.

Constraint Generalization. The function $generalize(C', P, Q, \Phi, C)$ computes a maximal set that contains C' , is contained within C , and only permits mappings that satisfy Φ . This function is used in two different ways: (1) to identify an undesirable region of the candidate space that should be avoided, and (2) to produce a maximal version of a valid mapping constraint.

The procedure works by incrementally growing C' into a larger set $C_{relaxed}$ and stopping when $C_{relaxed}$ contains at least one mapping that violates Φ . Suppose that constraint C' is represented by a symbolic expression \mathcal{X} , which itself is a conjunction of n subexpressions $k_1 \wedge k_2 \wedge \dots \wedge k_n$, where each k_i for $1 \leq i \leq n$ represents a (possibly conditional) assignment of a variable or a constant to some label parameter. The function $decompose(C')$ takes the given constraint and returns the set of such subexpressions. The function $relax(C', k_i)$ then computes a new constraint by removing k from C' ; this new constraint, $C_{relaxed}$, is a larger set of mappings that subsumes C' .

The verifier is then used to check $C_{relaxed}$ against Φ (line 22). If $C_{relaxed}$ is still valid with respect to Φ , then the implementation decision encoded by k is irrelevant for Φ , meaning we can safely remove k from the final synthesized constraint C' (line 24). If not, k is retained as part of C' , and the algorithm moves onto the next subexpression k as a candidate for removal (line 20). On line 23, we also make sure that $C_{relaxed}$ does not violate the predefined user constraints C .

Example. Let $\text{abs} = \text{a.e}.msg$ be a symbolic label that represents Alice sending a message (msg) to Eve, and $\text{conc} = msg'.key$ be its corresponding label in the public channel model. Then, one candidate constraint C' for mappings from the high-level to low-level labels can be specified as the following expression:

$$\mathcal{X}(\text{a.e}.msg, msg'.key) \equiv msg = msg' \wedge (msg = s \Rightarrow key = y) \wedge (msg = p \Rightarrow key = x)$$

Suppose that this constraint C' has been verified to be valid with respect to P , Q and Φ . Next, the generalization procedure removes the subexpression $k_1 \equiv (msg = p \Rightarrow key = x)$ from C' , resulting in constraint C_{relaxed} that is represented as:

$$\mathcal{X}(\text{a.e}.msg, msg'.key) \equiv msg = msg' \wedge (msg = s \Rightarrow key = y)$$

When checked by the verifier (line 22), C' is still considered valid, meaning that the decision encoded by k_1 is irrelevant to the property; thus, k_1 can be safely removed.

However, removing $k_2 \equiv (msg = s \Rightarrow key = y)$ results in a violation of the property. Thus, k_2 is kept as part of the final maximal constraint expression.

5 Implementation and Case Studies

5.1 Implementation

We have built a prototype implementation² of the synthesis algorithm described in Sect. 4. Our tool uses the Alloy Analyzer [25] as the underlying modeling and verification engine. Alloy's flexible, declarative relational logic is convenient for encoding the semantics of the mapping composition as well as specifying mapping constraints. The analysis engine for Alloy uses an off-the-shelf SAT solver to perform *bounded* verification [25]. In particular, our current prototype is capable of synthesizing mappings to preserve the following types of properties: *reachability* and *safety* properties, which can be expressed in either of the forms $\exists t : t \in T_P \wedge t \in \phi$ (reachability) and $\neg \exists t : t \in T_P \wedge t \notin \phi$ (safety) for some process P and property ϕ .

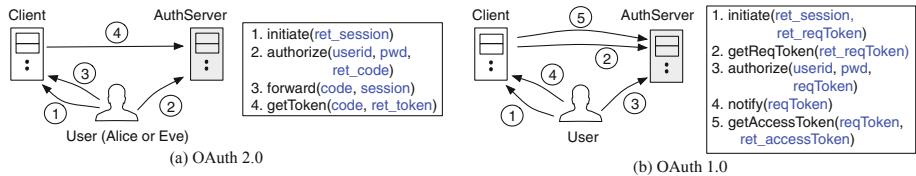


Fig. 3. A high-level overview of the two OAuth protocols, with a sequence of event labels that describe protocol steps in the typical order that they occur. Each arrowed edge indicates the direction of the communication. Variables inside labels with the prefix `ret_` represent return parameters. For example, in Step 2 of OAuth 2.0, User passes their user ID and password as arguments to AuthServer, which returns `ret_code` back to User in response.

² The tool, along with the models used in our case studies, is available at <https://github.com/eskang/MappingSynthesisTool>.

However, our synthesis approach does not prescribe the use of a particular modeling and verification engine, and can be implemented using other tools as well (such as an SMT solver [11, 12]).

5.2 Case Studies: OAuth Protocols

As two major case studies, we took on the problem of synthesizing valid mappings for *OAuth 1.0* and *OAuth 2.0*, two real-world protocols used for *third-party authorization* [24]. The purpose of the OAuth protocol family in general is to allow an application (called a *client* in the OAuth terminology) to access a resource from another application (an *authorization server*) without needing the credentials of the resource owner (a *user*). For example, a gaming application may initiate an OAuth process to obtain a list of friends from a particular user’s Facebook account, provided that the user has authorized Facebook to release this resource to the client.

OAuth 2.0 is the newer version of the protocol, while OAuth 1.0 is an older version. Although OAuth 2.0 is intended to be a replacement for OAuth 1.0, there has been much contention within the developer community about whether it actually improves over its predecessor in terms of security [17]. Since both protocols are designed to provide the same security guarantees (i.e., both share common properties), our goal was to apply our synthesis approach to systematically compare what developers would be required to do in order to construct secure web-based implementations of the two.

5.3 Formal Modeling

For our case studies, we constructed the following set of Alloy models: (1) model $P_{1.0}$ representing OAuth 1.0; (2) model $P_{2.0}$ representing OAuth 2.0; (3) model Q representing generic HTTP interactions between a browser and a server, as well as the behavior of a web-based attacker; (4) specification Φ describing desired protocol properties (same for both OAuth 1.0 and 2.0); and (5) mapping constraints $C_{1.0}$ and $C_{2.0}$ representing initial, user-specified partial mappings for OAuth 1.0 and 2.0, respectively. The complete models are approximately 1800 lines of Alloy code in total, and took around 4 man-months to build. These models were then provided as inputs to our tool to solve two instances of Problem 4 from Sect. 3. In particular, we synthesized a maximal mapping constraint $C'_{1.0}$ such that every $m \in C'_{1.0}$ ensures that $P_{1.0} \parallel_m Q \models \Phi$, and a maximal mapping constraint $C'_{2.0}$ such that every $m \in C'_{2.0}$ ensures that $P_{2.0} \parallel_m Q \models \Phi$.

OAuth Models ($P_{1.0}, P_{2.0}$). We constructed Alloy models of OAuth 1.0 and 2.0 based on the official protocol specifications [23, 24]. Due to limited space, we give only a brief overview of the models. Each model consists of four processes: Client, AuthServer, and two users, Alice and Eve (the latter with a malicious intent to access Alice’s resources).

A typical OAuth 2.0 workflow, shown in Fig. 3(a), begins with a user (Alice or Eve) initiating a new protocol session with Client (initiate). The user is then asked to prove their own identity to AuthServer (by providing a user ID and a password) and officially authorize the client to access their resources (authorize). Given the user’s authorization, the server then allocates a unique code for the user, and then redirects their back to the

client. The user forwards the code to the client (forward), which then can exchange the code for an access token to their resources (getToken).

Like in OAuth 2.0, a typical workflow in OAuth 1.0 (depicted in Fig. 3(b)) begins with a user initiating a new session with Client (initiate). Instead of immediately directing the user to AuthServer, however, Client first obtains a *request token* from AuthServer and associates it with the current session (getReqToken). The user is then asked to present the same request token to AuthServer and authorize Client to access their resources (authorize). Once notified by the user that the authorization step has taken place (notify), Client exchanges the request token for an access token that can be used subsequently to access their resources (getAccessToken).

Specification (Φ). There are two desirable properties of OAuth protocols in general: (1) **Authenticity**: When the client receives an access token, it must correspond to the user who initiated the current protocol session. (2) **Completion**: There exists at least one trace in which the protocol interactions are carried out to completion in the order of steps described in Fig. 3. Authenticity is a safety property while completion is a reachability property. The input specification Φ consists of these two properties. Completion is essential for ruling out mappings that over-constrain the resulting implementation and prevent certain steps of the protocol from being performed.

HTTP Platform Model (Q). Our goal was to explore and synthesize *web-based* implementations of OAuth. For this purpose, we constructed a formal model depicting interactions between a generic HTTP server and web browser. The model contains two types of processes, Server and Browser (which may be instantiated into multiple processes representing different servers and browsers). They interact with each other over HTTP requests, which share the following signature:

```
req(method : Method, url : URL, headers : List[Header], body : Body, ret_resp : Resp)
```

The parameters of an HTTP request have their own internal structures, each consisting of its own parameters as follows:

```
url(host : Host, path : Path, queries : List[Query]) header(name : Name, val : Value)
resp(status : Status, headers : List[Header], body : Body)
```

initiate(ret_session) \mapsto req(GET, http://client.com/initiate?queries, headers, body, ret_resp(OK, [set-cookie: ret_session], body))	forward(code, session) \mapsto req(POST, http://client.com/forward?queries, headers, body, ret_resp(OK, [], body))
authorize(userid, pwd, ret_code) \mapsto req(POST, http://server.com/authorize?queries, headers, body, ret_resp(Redirect, headers, body))	getToken(code, ret_token) \mapsto req(GET, http://client.com/getToken?[code], headers, body, ret_resp(OK, [], ret_token))

Fig. 4. User-specified partial mappings from OAuth 2.0 to HTTP. Terms highlighted in blue and red are variables that represent the parameters inside OAuth and HTTP labels, respectively. For example, in forward, the abstract parameters code and session may be transmitted as part of an URL query, a header, or the request body, although its URL is fixed to <http://client.com/forward>. (Color figure online)

Our model describes *generic*, *application-independent* HTTP interactions. In particular, each Browser process is a machine that constructs, at each communication step with Server, an arbitrary HTTP request by non-deterministically selecting a value for each parameter of the request. The processes, however, follow a *platform-specific* logic; for instance, when given a response from Server that instructs a browser cookie to be stored at a particular URL, Browser will include this cookie along with every subsequent request directed at that URL. In addition, the model includes a process that depicts the behavior of a web attacker, who may operate their own malicious server and exploit weaknesses in a browser to manipulate the user into sending certain HTTP requests.

Mapping Constraint ($C_{1.0}, C_{2.0}$). Building a web-based implementation of OAuth involves decisions about how abstract protocol operations are to be realized in terms of HTTP requests. As an input to the synthesizer, we specified an initial set of constraints that describe partial implementation decisions for both OAuth protocols; the ones for OAuth 2.0 are shown in Fig. 4. These decisions include a designation of fixed host and path names inside URLs for various OAuth operations (e.g., `http://client.com/initiate` for the OAuth `initiate` event), and how certain parameters are transmitted as part of an HTTP request (`ret_session` as a return cookie in `initiate`). It is reasonable to treat these constraints as given, since they describe decisions that are common across typical web-based OAuth implementations.

Insecure Mapping for OAuth 2.0. Let us now give an example of an insecure mapping that satisfies the user-given constraint in Fig. 4 but could introduce a security vulnerability into the resulting implementation. Later in Sect. 5.4, we describe how our tool can be used to synthesize a secure mapping that prevents this vulnerability.

Consider the OAuth 2.0 workflow from Fig. 3(a). In order to implement the `forward` operation, for instance, the developer must determine how the parameters `code` and `session` of the abstract event label are encoded using their concrete counterparts in an HTTP request. A number of choices is available. In one possible implementation, the authorization code may be transmitted as a query parameter inside the URL, and the session as a browser cookie, as described by the following constraint expression, \mathcal{X}_1 :

$$\begin{aligned} \mathcal{X}_1(a,b) \equiv & (b.method = \text{POST}) \wedge (b.url.host = \text{client.com}) \wedge \\ & (b.url.path = \text{forward}) \wedge (b.url.queries[0] = a.code) \wedge \\ & (b.headers[0].name = \text{cookie}) \wedge (b.headers[0].value = a.session) \end{aligned}$$

where `POST`, `client.com`, `forward`, and `cookie` are predefined constants; and $l[i]$ refers to i -th element of list l .

This constraint, however, allows a vulnerable implementation where malicious user Eve performs the first two steps of the workflow in Fig. 3(a) using her own credentials, and obtains a unique code ($code_{Eve}$) from the authorization server. Instead of forwarding this to Client (as she is expected to), Eve keeps the code herself, and crafts their own web page that triggers the visiting browser to send the following HTTP request:

```
req(POST, http://client.com/forward?codeEve, ...)
```

Suppose that Alice is a naive browser user who may occasionally be enticed or tricked into visiting malicious web sites. When Alice visits the page set up by Eve, Alice's

browser automatically generates the above HTTP request, which, given the decisions in \mathcal{X}_1 , corresponds to a valid forward event:

```
forward(codeEve, sessionAlice) ↪
req(POST, http://client.com/forward?codeEve, [(cookie, sessionAlice)], ...)
```

Due to the standard browser logic, the cookie corresponding to session_{Alice} is included in every request to client.com. As a result, Client mistakenly accepts code_{Eve} as the one for Alice, even though it belongs to Eve, violating the authenticity property of OAuth (this attack is also called *session swapping* [39]).

5.4 Results

Our synthesis tool was able to generate valid mapping constraints for both OAuth protocols. In particular, the constraints describe mitigations against attacks that exploit an interaction between the OAuth logic and security vulnerabilities in a web browser.

OAuth 2.0. The synthesized symbolic mapping constraint for OAuth 2.0 consists of 39 conjuncts in total, each capturing a (conditional) assignment of a concrete HTTP parameter to a constant (e.g., $b.url.path = \text{forward}$) or an abstract OAuth parameter (e.g., $b.url.queries[0] = a.code$). In particular, the constraint captures mitigations against *session swapping* [39] and *covert redirect* [16]. Due to limited space, we omit the full constraint, but instead describe how the vulnerability described at the end of Sect. 5.3 can be mitigated by our synthesized mapping.

Consider the insecure mapping expression \mathcal{X}_1 from Sect. 5.3. The mapping constraint synthesized by our tool, \mathcal{X}_2 , fixes the major problem of \mathcal{X}_1 ; namely, that in a browser-based implementation, the client cannot trust an authorization code as having originated from a particular user (e.g., Alice), since the code may be intercepted or interjected by an attacker (Eve) while in transit through a browser. A possible solution is to explicitly identify the origin of the code by requiring an additional piece of tracking information to be provided in each forward request. The mapping expression \mathcal{X}_2 synthesized by our tool encodes one form of this solution:

$$\begin{aligned} \mathcal{X}_2(a, b) \equiv & \mathcal{X}_1(a, b) \wedge (a.session = \text{session}_{\text{Alice}} \Rightarrow b.url.queries[1] = \text{nonce}_0) \wedge \\ & (a.session = \text{session}_{\text{Eve}} \Rightarrow b.url.queries[1] = \text{nonce}_1) \end{aligned}$$

where $\text{nonce}_0, \text{nonce}_1 \in \text{Nonce}$ are constants defined in the HTTP model³. In particular, \mathcal{X}_2 stipulates that every forward request must include an additional value (nonce) as an argument besides the code and the session, and that this nonce be unique for each session value. \mathcal{X}_2 ensures that the resulting implementation satisfies the desired properties of OAuth 2.

OAuth 1.0. The synthesized symbolic mapping constraint for OAuth 1.0 consists of 48 conjuncts in total, capturing how the abstract parameters of the five OAuth 1.0 operations are related to concrete HTTP parameters. The constraint synthesized by our tool

³ A nonce is a unique piece of string intended to be used once in communication.

	# total candidates	# explored	# verified	# skipped	Verification		Generalization	Total time
					Avg.	Total		
OAuth 1.0	79200	2465	281	2184	2.01	566.05	490.84	1056.89
OAuth 2.0	29400	1453	161	1292	1.88	302.76	1138.85	1441.60

Fig. 5. Experimental results (all times in seconds). “# total candidates” is the total number of possible symbolic mapping expressions; “# explored” is the number of iterations taken by the main synthesis loop (lines 3–15, Fig. 2) before a solution was found. Out of these iterations, “# verified” mappings were verified (line 7), while the rest were identified as invalid and skipped (line 5). “Total time” the sum of the Total Verification and Generalization columns) refers to the time spent by the tool to synthesize a maximal constraint.

for OAuth 1.0 encodes a mitigation against the *session fixation* [15] attack; in short, this mitigation involves strengthening the *notify* operation with unique nonces (similar to the way the *forward* operation in OAuth 2.0 was fixed above) to prevent the attacker from violating the authenticity property.

Performance. Figure 5 shows experimental results for the two OAuth protocols⁴. Overall, the synthesizer took approximately 17.6 and 24.0 min to synthesize the constraints for 1.0 and 2.0, respectively. In both cases, the tool spent a considerable amount of time on the generalization step to learn the invalid regions of the search space. Note that generalization is effective at identifying and discarding a very large number of invalid candidates; it was able to skip 2184 out of 2465 candidates for OAuth 1.0 ($\approx 88.6\%$) and 1292 out of 1453 for OAuth 2.0 ($\approx 88.9\%$). Our generalization technique was particularly effective for the OAuth protocols, since a significant percentage of the candidate constraints would result in an implementation that violates the completion property (i.e., it prevents Alice or Eve from completing a protocol session in an expected order). Often, the decisions contributing to this violation could be localized to a small subset of entries in a mapping (for example, attempting to send a cookie to a mismatched URL, which is inconsistent with the behavior of the browser process). By identifying this subset, our algorithm was able to discover and eliminate a large number of invalid mappings.

6 Related Work

Our approach has been inspired by the success of recent synthesis paradigms such as *sketching* [36–38], *oracle-guided synthesis* [26] and *syntax-guided synthesis* [3]. Our technique shares many similarities with these approaches in that (1) it allows the user to provide a partial specification of the artifact to be synthesized (in the form of constraints or examples), therefore having the underlying engine *complete* the remaining parts; (2) it relies on an interaction between the verifier, which checks candidate solutions, and the synthesizer, which prunes that search space based on previous invalid candidates. Our work also differs in a number of aspects. First, we synthesize mappings from high-level models to low-level execution platforms, which to our knowledge has not been

⁴ The experiments were performed on a Mac OS X 2.7 GHz laptop with 8G RAM and Minisat [13] as the underlying SAT solver employed by the Alloy Analyzer.

considered before. Second, our approach leverages constraint generalization to not only prune the search space, but also to produce a constraint capturing a (locally) maximal set of valid mappings. Third, our application domain is in security protocols.

A large body of literature exists on *refinement-based* methods to system construction [4, 20]. These approaches involve building an implementation Q that is a behavioral refinement of P ; such Q , by construction, would satisfy the properties of P . In comparison, we start with an assumption that Q is a *given* platform, and that the developer may not have the luxury of being able to modify or build Q from scratch. Thus, instead of behavioral refinement (which may be too challenging to achieve), we aim to preserve some critical property ϕ when P is implemented using Q .

The task of synthesizing a valid mapping can be seen as a type of the *model merging* problem [8]. This problem has been studied in various contexts, including architectural views [31], behavioral models [6, 32, 40], and database schemas [34]. Among these, our work is most closely related to merging of partial behavioral models [6, 40]. In these works, given a pair of models M_1 and M_2 , the goal is to construct M' that is a behavioral refinement of both M_1 and M_2 . The approach proposed in this paper differs in that (1) the mapping composition involves merging a pair of events with distinct alphabet labels into a single event that retains all of those labels, and (2) the composed process ($P \parallel_m Q$) need not be a behavioral refinement of P or Q , as long as it satisfies property ϕ .

Bhargavan and his colleagues presents a compiler that takes a high-level program written using *session types* [22] and automatically generates a low-level implementation [7]. This technique is closer to compilation than to synthesis in that it uses a fixed translation scheme from high-level to low-level operations in a specific language environment (.NET), without searching a space of possible translations. Synthesizing a low-level implementation from a high-level specification has also been studied in the context of data structures [18, 19], although their underlying representation (relational algebra for data schema specification) is very different from ours (process algebra).

A significant contribution of our work is the production of formal models for real-world protocols such as OAuth and HTTP. There have been similar efforts by other researchers in building reusable models of the web for security analysis [1, 5, 14]. As far as we know, however, none of these models has been used for synthesis.

7 Conclusions

In this paper, we have proposed a novel system design methodology centered around the notion of *mappings*. We have presented novel *mapping synthesis problems* and an algorithm for efficiently synthesizing symbolic maximal valid mappings. In addition, we have validated our approach on realistic case studies involving the OAuth protocols.

Future directions include performance improvements (e.g., exploiting the fact that our generalization-based algorithm is easily parallelizable), combining our generalization-based synthesis method with a counter-example guided approach, and application of our synthesis approach to other domains beside security (e.g., platform-based design and embedded systems [35]).

References

1. Akhawe, D., Barth, A., Lam, P.E., Mitchell, J.C., Song, D.: Towards a formal foundation of web security. In: CSF, pp. 290–304 (2010)
2. Alpern, B., Schneider, F.B.: Defining liveness. Inf. Process. Lett. **21**(4), 181–185 (1985)
3. Alur, R., et al.: Syntax-guided synthesis. In: FMCAD, pp. 1–8 (2013)
4. Back, R.-J.: A calculus of refinements for program derivations. Acta Inf. **25**(6), 593–624 (1988)
5. Bansal, C., Bhargavan, K., Maffeis, S.: Discovering concrete attacks on website authorization by formal analysis. In: CSF, pp. 247–262 (2012)
6. Ben-David, S., Chechik, M., Uchitel, S.: Merging partial behaviour models with different vocabularies. In: CONCUR, pp. 91–105 (2013)
7. Bhargavan, K., Corin, R., Deniéou, P.-M., Fournet, C., Leifer, J.J.: Cryptographic protocol synthesis and verification for multiparty sessions. In: CSF, pp. 124–140 (2009)
8. Brunet, G., Chechik, M., Easterbrook, S., Nejati, S., Niu, N., Sabetzadeh, M.: A manifesto for model merging. In: Proceedings of the 2006 International Workshop on Global Integrated Model Management, pp. 5–12 (2006)
9. Chari, S., Jutla, C.S., Roy, A.: Universally Composable Security Analysis of OAuth v2.0. IACR Cryptology ePrint Archive, 2011, 526 (2011)
10. Chen, E.Y., Pei, Y., Chen, S., Tian, Y., Kotcher, R., Tague, P.: OAuth Demystified for Mobile Application Developers. In: CCS, pp. 892–903 (2014)
11. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
12. Dutertre, B.: Yices 2.2. In: CAV, pp. 737–744 (2014)
13. Eén, N., Sörensson, N.: An extensible sat-solver. In: SAT, pp. 502–518 (2003)
14. Fett, D., Küsters, R., Schmitz, G.: An expressive model for the web infrastructure: definition and application to the browser ID SSO system. In: 2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, 18–21 May 2014, pp. 673–688 (2014)
15. OAuth Working Group. OAuth Security Advisory: 2009.1 “Session Fixation”. <https://oauth.net/advisories/2009-1> (2009)
16. OAuth Working Group. OAuth Security Advisory: 2014.1 “Covert Redirect”. <https://oauth.net/advisories/2014-1-covert-redirect> (2014)
17. Hanmer, E.: OAuth 2.0 and the Road to Hell. <https://hueniverse.com/2012/07/26/oauth-2-0-and-the-road-to-hell> (2012)
18. Hawkins, P., Aiken, A., Fisher, K., Rinard, M.C., Sagiv, M.: Data representation synthesis. In: PLDI, pp. 38–49 (2011)
19. Hawkins, P., Aiken, A., Fisher, K., Rinard, M.C., Sagiv, M.: Concurrent data representation synthesis. In: PLDI, pp. 417–428 (2012)
20. Hoare, C.A.R.: Proof of correctness of data representations. Acta Inf. **1**, 271–281 (1972)
21. Hoare, C.A.R.: Communicating sequential processes. Commun. ACM **21**(8), 666–677 (1978)
22. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: POPL, pp. 273–284 (2008)
23. Internet Engineering Task Force. The OAuth 1.0 Protocol. <https://tools.ietf.org/html/rfc5849> (2010)
24. Internet Engineering Task Force. OAuth Authorization Framework. <http://tools.ietf.org/html/rfc6749> (2014)
25. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press, Cambridge (2006)

26. Jha, S., Gulwani, S., Seshia, S.A., Tiwari, A.: Oracle-guided component-based program synthesis. In: ICSE, pp. 215–224 (2010)
27. Kang, E., Lafourture, S., Tripakis, S.: Synthesis of property-preserving mappings. CoRR, abs/1705.03618 (2017)
28. Kang, E., Milicevic, A., Jackson, D.: Multi-representational security analysis. In: FSE (2016)
29. Lamport, L.: Proving the correctness of multiprocess programs. IEEE Trans. Software Eng. **3**(2), 125–143 (1977)
30. Lampson, B.W.: A note on the confinement problem. Commun. ACM **16**(10), 613–615 (1973)
31. Maoz, S., Ringert, J.O., Rumpe, B.: Synthesis of component and connector models from crosscutting structural views. In: ESEC/FSE, pp. 444–454 (2013)
32. Nejati, S., Sabetzadeh, M., Chechik, M., Easterbrook, S.M., Zave, P.: Matching and merging of statecharts specifications. In: ICSE, pp. 54–64 (2007)
33. Pai, S., Sharma, Y., Kumar, S., Pai, R.M., Singh, S.: Formal verification of OAuth 2.0 using Alloy framework. In: Communication Systems and Network Technologies (CSNT), pp. 655–659. IEEE (2011)
34. Pottinger, R., Bernstein, P.A.: Merging models based on given correspondences. In: VLDB, pp. 826–873 (2003)
35. Sangiovanni-Vincentelli, A.L., Martin, G.: Platform-based design and software design methodology for embedded systems. IEEE Design Test Comput. **18**(6), 23–33 (2001)
36. Solar-Lezama, A.: The sketching approach to program synthesis. In: Hu, Z. (ed.) APLAS 2009. LNCS, vol. 5904, pp. 4–13. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-10672-9_3
37. Solar-Lezama, A., Rabbah, R., Bodík, R., Ebcioğlu, K.: Programming by sketching for bit-streaming programs. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2005, pp. 281–294. ACM (2005)
38. Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S.A., Saraswat, V.A.: Combinatorial sketching for finite programs. In: ASPLOS, pp. 404–415 (2006)
39. Sun, S.-T., Beznosov, K.: The devil is in the (implementation) details: an empirical analysis of OAuth SSO systems. In: CCS, pp. 378–390 (2012)
40. Uchitel, S., Chechik, M.: Merging partial behavioural models. In: SIGSOFT FSE, pp. 43–52 (2004)
41. Wang, R., Chen, S., Wang, X.: Signing me onto your accounts through facebook and google: a traffic-guided security study of commercially deployed single-sign-on web services. In: IEEE Symposium on Security and Privacy, pp. 365–379 (2012)
42. Xu, X., Niu, L., Meng, B.: Automatic verification of security properties of OAuth 2.0 protocol with cryptoverif in computational model. Inf. Technol. J. **12**(12), 2273 (2013)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.



Synthesis



Synthesizing Approximate Implementations for Unrealizable Specifications

Rayna Dimitrova¹, Bernd Finkbeiner², and Hazem Torfah^{2(✉)}

¹ University of Leicester, Leicester, UK

² Saarland University, Saarbrücken, Germany

torfah@react.uni-saarland.de

Abstract. The unrealizability of a specification is often due to the assumption that the behavior of the environment is unrestricted. In this paper, we present algorithms for synthesis in bounded environments, where the environment can only generate input sequences that are ultimately periodic words (lassos) with finite representations of bounded size. We provide automata-theoretic and symbolic approaches for solving this synthesis problem, and also study the synthesis of approximative implementations from unrealizable specifications. Such implementations may violate the specification in general, but are guaranteed to satisfy the specification on at least a specified portion of the bounded-size lassos. We evaluate the algorithms on different arbiter specifications.

1 Introduction

The objective of reactive synthesis is to automatically construct an implementation of a reactive system from a high-level specification of its desired behaviour. While this idea holds a great promise, applying synthesis in practice often faces significant challenges. One of the main hurdles is that the system designer has to provide the right formal specification, which is often a difficult task [12]. In particular, since the system being synthesized is required to satisfy its requirements against all possible environments allowed by the specification, accurately capturing the designer’s knowledge about the environment in which the system will execute is crucial for being able to successfully synthesize an implementation.

Traditionally, environment assumptions are included in the specification, usually given as a temporal logic formula. There are, however less explored ways of incorporating information about the environment, one of which is to consider a *bound on the size of the environment*, that is, a bound on the size of the state space of a transition system that describes the possible environment behaviours. Restricting the space of possible environments can render an unrealizable specification into a realizable one. The temporal synthesis under such

This work was partially supported by the German Research Foundation (DFG) as part of the Collaborative Research Center “Foundations of Perspicuous Software Systems” (TRR 248, 389792660), and by the European Research Council (ERC) Grant OSARES (No. 683300).

© The Author(s) 2019

I. Dillig and S. Tasiran (Eds.): CAV 2019, LNCS 11561, pp. 241–258, 2019.

https://doi.org/10.1007/978-3-030-25540-4_13

bounded environments was first studied in [6], where the authors extensively study the problem, in several versions, from the complexity-theoretic point of view.

In this paper, we follow a similar avenue of providing environment assumptions. However, instead of bounding the size of the state space of the environment, we associate a bound with the sequences of values of input signals produced by the environment. The infinite input sequences produced by a finite-state environment which interacts with a finite state system are ultimately periodic, and thus, each such infinite sequence $\sigma \in \Sigma_I^\omega$, over the input alphabet Σ_I , can be represented as a *lasso*, which is a pair (u, v) of finite words $u \in \Sigma_I^*$ and $v \in \Sigma_I^+$, such that $\sigma = u \cdot v^\omega$. It is the length of such sequences that we consider a bound on. More precisely, given a bound $k \in \mathbb{N}$, we consider the language of all infinite sequences of inputs that can be represented by a lasso (u, v) with $|u \cdot v| = k$. The goal of the *synthesis of lasso precise implementations* is then to synthesize a system for which each execution resulting from a sequence of environment inputs in that language, satisfies a given linear temporal specification.

As an example, consider an arbiter serving two client processes. Each client issues a request when it wants to access a shared resource, and keeps the request signal up until it is done using the resource. The goal of the arbiter is to ensure the classical mutual exclusion property, by not granting access to the two clients simultaneously. The arbiter has to also ensure that each client request is eventually granted. This, however, is difficult since, first, a client might gain access to the resource and never lower the request signal, and second, the arbiter is not allowed to take away a grant unless the request has been set to false, or the client never sets the request to false in the future (the client has become unresponsive). The last two requirements together make the specification unrealizable, as the arbiter has no way of determining if a client has become unresponsive, or will lower the request signal in the future. If, however, the length of the lassos of the input sequences is bounded, then, after a sufficient number of steps, the arbiter can assume that if the request has not been set to false, then it will not be lowered in the future either, as the sequence of inputs must already have run at least once through its period that will be ultimately repeated from that point on.

Formally, we can express the requirements on the arbiter in Linear Temporal Logic (LTL) as follows. There is one input variable r_i (for *request*) and one output variable g_i (for *grant*) associated with each client. The specification is then given as the conjunction $\varphi = \varphi_{mutex} \wedge \varphi_{resp} \wedge \varphi_{rel}$ where we use the LTL operators Next \bigcirc , Globally \square and Eventually \diamond to define the requirements

$$\begin{aligned}\varphi_{mutex} &= \square \neg(g_1 \wedge g_2), \\ \varphi_{resp} &= \square \bigwedge_{i=1}^2 (r_i \rightarrow \diamond g_i), \\ \varphi_{rel} &= \square \bigwedge_{i=1}^2 (g_i \wedge r_i \wedge (\diamond \neg r_i) \rightarrow \bigcirc g_i).\end{aligned}$$

Due to the requirement to not revoke grants stated in φ_{rel} , the specification φ is unrealizable (that is, there exists no implementation for the arbiter process). For any bound k on the length of the input lassos, however, φ is realizable. More precisely, there exists an implementation in which once client i has not lowered the request signal for k consecutive steps, the variable g_i is set to false.

This example shows that when the system designer has knowledge about the resources available to the environment processes, taking this knowledge into account can enable us to synthesize a system that is correct under this assumption.

In this paper we formally define the synthesis problem for *lasso-precise implementations*, that is, implementations that are correct for input lassos of bounded size, and describe an automata-theoretic approach to this synthesis problem. We also consider the synthesis of *lasso-precise implementations of bounded size*, and provide a symbolic synthesis algorithm based on quantified Boolean satisfiability.

Bounding the size of the input lassos can render some unrealizable specifications realizable, but, similarly to bounding the size of the environment, comes at the price of higher computational complexity. To alleviate this problem, we further study the synthesis of *approximate implementations*, where we relax the synthesis problem further, and only require that for a given $\epsilon > 0$ the ratio of input lassos of a given size for which the specification is satisfied, to the total number of input lassos of that size is at least $1 - \epsilon$. We then propose an *approximate synthesis method* based on maximum model counting for Boolean formulas [5]. The benefits of the approximate approach are two-fold. Firstly, it can often deliver high-quality approximate solutions more efficiently than the lasso-precise synthesis method, and secondly, even when the specification is still unrealizable for a given lasso bound, we might be able to synthesize an implementation that is correct for a given fraction of the possible input lassos.

The rest of the paper is organized as follows. In Sect. 2 we discuss related work on environment assumptions in synthesis. In Sect. 3 we provide preliminaries on linear temporal properties and omega-automata. In Sect. 3 we define the synthesis problem for lasso-precise implementations, and describe an automata-theoretic synthesis algorithm. In Sect. 5 we study the synthesis of lasso-precise implementations of bounded size, and provide a reduction to quantified Boolean satisfiability. In Sect. 6 we define the approximate version of the problem, and give a synthesis procedure based on maximum model counting. Finally, in Sect. 7 we present experimental results, and conclude in Sect. 8.

2 Related Work

Providing good-quality environment specifications (typically in the form of assumptions on the allowed behaviours of the environment) is crucial for the synthesis of implementations from high-level specifications. Formal specifications, and thus also environment assumptions, are often hard to get right, and have been identified as one of the bottlenecks in formal methods and autonomy [12]. It is therefore not surprising, that there is a plethora of approaches addressing

the problem of how to revise inadequate environment assumptions in the cases when these are the cause of unrealizability of the system requirements.

Most approaches in this direction build upon the idea of analyzing the cause of unrealizability of the specification and extracting assumptions that help eliminate this cause. The method proposed in [2] uses the game graph that is used to answer the realizability question in order to construct a Büchi automaton representing a minimal assumption that makes the specification realizable. The authors of [8] provide an alternative approach where the environment assumptions are gradually strengthened based on counterstrategies for the environment. The key ingredient for this approach is using a library of specification templates and user scenarios for the mining of assumptions, in order to generate good-quality assumptions. A similar approach is used in [1], where, however, assumption patterns are synthesized directly from the counterstrategy without the need for the user to provide patterns. A different line of work focuses on giving feedback to the user or specification designer about the reason for unrealizability, so that they can, if possible, revise the specification accordingly. The key challenge addressed there lies in providing easy-to-understand feedback to users, which relies on finding a minimal cause for why the requirements are not achievable and generating a natural language explanation of this cause [11].

In the above mentioned approaches, assumptions are provided or constructed in the form of a temporal logic formula or an omega-automaton. Thus, it is on the one hand often difficult for specification designers to specify the right assumptions, and on the other hand special care has to be taken by the assumption generation procedures to ensure that the constructed assumptions are simple enough for the user to understand and evaluate. The work [6] takes a different route, by making assumptions about the *size* of the environment. That is, including as an additional parameter to the synthesis problem a bound on the state space of the environment. Similarly to temporal logic assumptions, this relaxation of the synthesis problem can render unrealizable specifications into realizable ones. From the system designer point of view, however, it might be significantly easier to estimate the size of environments that are feasible in practice than to express the implications of this additional information in a temporal logic formula. In this paper we take a similar route to [6], and consider a bound on the cyclic structures in the environment's behaviour. Thus, the closest to our work is the temporal synthesis for bounded environments studied in [6]. In fact, we show that the synthesis problem for lasso-precise implementations and the synthesis problem under bounded environments can be reduced to each other. However, while the focus in [6] is on the computational complexity of the bounded synthesis problems, here we provide both automata-theoretic, as well as symbolic approaches for solving the synthesis problem for environments with bounded lassos. We further consider an *approximate version of this synthesis problem*. The benefits of using approximation are two-fold. Firstly, as shown in [6], while bounding the environment can make some specifications realizable, this comes at a high computational complexity price. In this case, approximation might be able to provide solutions of sufficient quality more efficiently. Furthermore,

even after bounding the environment's input behaviours, the specification might still remain unrealizable, in which case we would like to satisfy the requirements for as many input lassos as possible. In that sense, we get closer to synthesis methods for probabilistic temporal properties in probabilistic environments [7]. However, we consider non-probabilistic environments (i.e., all possible inputs are equally likely), and provide probabilistic guarantees with desired confidence by employing maximum model counting techniques. Maximum model counting has previously been used for the synthesis of approximate non-reactive programs [5]. Here, on the other hand we are concerned with the synthesis of reactive systems from temporal specifications.

Bounding the size of the synthesized system implementation is a complementary restriction of the synthesis problem, which has attracted a lot of attention in recent years [4]. The computational complexity of the synthesis problem when both the system's and the environment's size is bounded has been studied in [6]. In this paper we provide a symbolic synthesis procedure for bounded synthesis of lasso-precise implementations based on quantified Boolean satisfiability.

3 Preliminaries

We now recall definitions and notation from formal languages and automata, and notions from reactive synthesis such as implementation and environment.

Linear-Time Properties and Lassos. A *linear-time property* φ over an alphabet Σ is a set of infinite words $\varphi \subseteq \Sigma^\omega$. Elements of φ are called *models* of φ . A *lasso* of length k over an alphabet Σ is a pair (u, v) of finite words $u \in \Sigma^*$ and $v \in \Sigma^+$ with $|u \cdot v| = k$ that induces the ultimately periodic word $u \cdot v^\omega$. We call $u \cdot v$ the *base* of the lasso or ultimately periodic word, and k the *length* of the lasso.

If a word $w \in \Sigma^*$ is a prefix of a word $\sigma \in \Sigma^* \cup \Sigma^\omega$, we write $w < \sigma$. For a language $L \subseteq \Sigma^* \cup \Sigma^\omega$, we define $\text{Prefix}(L) = \{w \in \Sigma^* \mid \exists \sigma \in L : w < \sigma\}$ is the set of all finite words that are prefixes of words in L .

Implementations. We represent implementations as *labeled transition systems*. Let I and O be finite sets of *input* and *output atomic propositions* respectively. A 2^O -labeled 2^I -transition system is a tuple $\mathcal{T} = (T, t_0, \tau, o)$, consisting of a finite set of states T , an initial state $t_0 \in T$, a transition function $\tau: T \times 2^I \rightarrow T$, and a labeling function $o: T \rightarrow 2^O$. We denote by $|\mathcal{T}|$ the size of an implementation \mathcal{T} , defined as $|\mathcal{T}| = |T|$. A *path* in \mathcal{T} is a sequence $\pi: \mathbb{N} \rightarrow T \times 2^I$ of states and inputs that follows the transition function, i.e., for all $i \in \mathbb{N}$ if $\pi(i) = (t_i, e_i)$ and $\pi(i+1) = (t_{i+1}, e_{i+1})$, then $t_{i+1} = \tau(t_i, e_i)$. We call a path *initial* if it starts with the initial state: $\pi(0) = (t_0, e)$ for some $e \in 2^I$. For an initial path π , we call the sequence $\sigma_\pi: i \mapsto (o(t_i) \cup e_i) \in (2^{I \cup O})^\omega$ the *trace* of π . We call the set of traces of a transition system \mathcal{T} the *language* of \mathcal{T} , denoted $L(\mathcal{T})$.

Finite-state environments can be represented as labelled transition systems in a similar way, with the difference that the inputs are the outputs of the implementation, and the states of the environment are labelled with inputs for

the implementation. More precisely, a finite-state environment is a 2^I -labeled 2^O -transition system $\mathcal{E} = (E, s_0, \rho, \iota)$. The composition of an implementation \mathcal{T} and an environment \mathcal{E} results in a set of traces of \mathcal{T} , which we denote $L_{\mathcal{E}}(\mathcal{T})$, where $\sigma = \sigma_0\sigma_1\dots \in L_{\mathcal{E}}(\mathcal{T})$ if and only if $\sigma \in L(\mathcal{T})$ and there exists an initial path $s_0s_1\dots$ in \mathcal{E} such that for all $i \in \mathbb{N}$, $s_{i+1} = \rho(s_i, \sigma_{i+1} \cap O)$ and $\sigma_i \cap I = \iota(s_i)$.

Linear-Time Temporal Logic. We specify properties of reactive systems (implementations) as formulas in Linear-time Temporal Logic (LTL) [9]. We consider the usual temporal operators Next \bigcirc , Until \mathcal{U} , and the derived operators Release \mathcal{R} , which is the dual operator of \mathcal{U} , Eventually \lozenge and Globally \square . LTL formulas are defined over a set of atomic propositions AP . We denote the satisfaction of an LTL formula φ by an infinite sequence $\sigma \in (2^{AP})^\omega$ of valuations of the atomic propositions by $\sigma \models \varphi$ and call σ a *model* of φ . For an LTL formula φ we define the language $L(\varphi)$ of φ to be the set $\{\sigma \in (2^{AP})^\omega \mid \sigma \models \varphi\}$.

For a set of atomic propositions $AP = O \cup I$, we say that a 2^O -labeled 2^I -transition system \mathcal{T} satisfies an LTL formula φ , if and only if $L(\mathcal{T}) \subseteq L(\varphi)$, i.e., every trace of \mathcal{T} satisfies φ . In this case we call \mathcal{T} a *model* of φ , denoted $\mathcal{T} \models \varphi$. If \mathcal{T} satisfies φ for an environment \mathcal{E} , i.e. $L_{\mathcal{E}}(\mathcal{T}) \subseteq L(\varphi)$, we write $\mathcal{T} \models_{\mathcal{E}} \varphi$.

For $I \subseteq AP$ and $\sigma \in (2^{AP})^* \cup (2^{AP})^\omega$, we denote with $\sigma|_I$ the projection of σ on I , obtained by the sequence of valuations of the propositions from I in σ .

Automata Over Infinite Words. The automata-theoretic approach to reactive synthesis relies on the fact that an LTL specification can be translated to an automaton over infinite words, or, alternatively, that the specification can be provided directly as such an automaton. An *alternating parity automaton* over an alphabet Σ is a tuple $\mathcal{A} = (Q, q_0, \delta, \mu)$, where Q denotes a finite set of states, $Q_0 \subseteq Q$ denotes a set of initial states, δ denotes a transition function, and $\mu : Q \rightarrow C \subset \mathbb{N}$ is a coloring function. The transition function $\delta : Q \times \Sigma \rightarrow \mathbb{B}^+(Q)$ maps a state and an input letter to a positive Boolean combination of states [14].

A tree T over a set of directions D is a prefix-closed subset of D^* . The empty sequence ϵ is called the root. The children of a node $n \in T$ are the nodes $\{n \cdot d \in T \mid d \in D\}$. A Σ -labeled tree is a pair (T, l) , where $l : T \rightarrow \Sigma$ is the labeling function. A *run* of $\mathcal{A} = (Q, q_0, \delta, \mu)$ on an infinite word $\sigma = \alpha_0\alpha_1\dots \in \Sigma^\omega$ is a Q -labeled tree (T, l) that satisfies the following constraints: (1) $l(\epsilon) = q_0$, and (2) for all $n \in T$, if $l(n) = q$, then $\{l(n') \mid n'$ is a child of $n\}$ satisfies $\delta(q, \alpha_{|n|})$.

A run tree is *accepting* if every branch either hits a *true* transition or is an infinite branch $n_0n_1n_2\dots \in T$, and the sequence $l(n_0)l(n_1)l(n_2)\dots$ satisfies the *parity condition*, which requires that the highest color occurring infinitely often in the sequence $\mu(l(n_0))\mu(l(n_1))\mu(l(n_2))\dots \in \mathbb{N}^\omega$ is even. An infinite word σ is accepted by an automaton \mathcal{A} if there exists an accepting run of \mathcal{A} on σ . The set of infinite words accepted by \mathcal{A} is called its *language*, denoted $L(\mathcal{A})$.

A *nondeterministic automaton* is a special alternating automaton, where for all states q and input letters α , $\delta(q, \alpha)$ is a disjunction. An alternating automaton is called *universal* if, for all states q and input letters α , $\delta(q, \alpha)$ is a conjunction. A universal and nondeterministic automaton is called *deterministic*.

A parity automaton is called a *Büchi* automaton if and only if the image of μ is contained in $\{1, 2\}$, a *co-Büchi* automaton if and only if the image of α is contained in $\{0, 1\}$. Büchi and co-Büchi automata are denoted by (Q, Q_0, δ, F) , where $F \subseteq Q$ denotes the states with the higher color. A run graph of a Büchi automaton is thus accepting if, on every infinite path, there are infinitely many visits to states in F ; a run graph of a co-Büchi automaton is accepting if, on every path, there are only finitely many visits to states in F .

The next theorem states the relation between LTL and alternating Büchi automata, namely that every LTL formula φ can be translated to an alternating Büchi automaton with the same language and size linear in the length of φ .

Theorem 1. [13] *For every LTL formula φ there is an alternating Büchi automaton \mathcal{A} of size $O(|\varphi|)$ with $L(\mathcal{A}) = L(\varphi)$, where $|\varphi|$ is the length of φ .*

Automata Over Finite Words. We also use automata over finite words as acceptors for languages consisting of prefixes of traces. A nondeterministic finite automaton over an alphabet Σ is a tuple $\mathcal{A} = (Q, Q_0, \delta, F)$, where Q and $Q_0 \subseteq Q$ are again the states and initial states respectively, $\delta : Q \times \Sigma \rightarrow 2^Q$ is the transition function and F is the set of accepting states. A run on a word $a_1 \dots a_n$ is a sequence of states $q_0 q_1 \dots q_n$, where $q_0 \in Q_0$ and $q_{i+1} \in \delta(q_i, a_i)$. The run is accepting if $q_n \in F$. Deterministic finite automata are defined similarly with the difference that there is a single initial state q_0 , and that the transition function is of the form $\delta : Q \times \Sigma \rightarrow Q$. As usual, we denote the set of words accepted by a nondeterministic or deterministic finite automaton \mathcal{A} by $L(\mathcal{A})$.

4 Synthesis of Lasso-Precise Implementations

In this section we first define the synthesis problem for environments producing input sequences representable as lassos of length bounded by a given number. We then provide an automata-theoretic algorithm for this synthesis problem.

4.1 Lasso-Precise Implementations

We begin by formally defining the language of sequences of input values representable by lassos of a given length k . For the rest of the section, we consider linear-time properties defined over a set of atomic propositions AP . The subset $I \subseteq AP$ consists of the input atomic propositions controlled by the environment.

Definition 1 (Bounded Model Languages). *Let φ be a linear-time property over a set of atomic propositions AP , let $\Sigma = 2^{AP}$, and let $I \subseteq AP$.*

We say that an infinite word $\sigma \in \Sigma^\omega$ is an I - k -model of φ , for a bound $k \in \mathbb{N}$, if and only if there are words $u \in (2^I)^$ and $v \in (2^I)^+$ such that $|u \cdot v| = k$ and $\sigma|_I = u \cdot v^\omega$. The language of I - k -models of the property φ is defined by the set $L_k^I(\varphi) = \{\sigma \in \Sigma^\omega \mid \sigma \text{ is a } I\text{-}k\text{-model of } \varphi\}$.*

Note that a model of φ might be induced by lassos of different length and by more than one lasso of the same length, e.g., a^ω is induced by (a, a) and (ϵ, aa) . The next lemma establishes that if a model of φ can be represented by a lasso of length k then it can also be represented by a lasso of any larger length.

Lemma 1. *For a linear-time property φ over $\Sigma = 2^{AP}$, subset $I \subseteq AP$ of atomic propositions, and bound $k \in \mathbb{N}$, we have $L_k^I(\varphi) \subseteq L_{k'}^I(\varphi)$ for all $k' > k$.*

Proof. Let $\sigma \in L_k^I(\varphi)$. Then, $\sigma \models \varphi$ and there exists $(u, v) \in (2^I)^* \times (2^I)^+$ such that $|u \cdot v| = k$ and $\sigma|_I = u \cdot v^\omega$. Let $v = v_1 \dots v_k$. Since $u \cdot v_1(v_2 \dots v_k v_1)^\omega = u \cdot (v_1 \dots v_k)^\omega = \sigma|_I$, we have $\sigma \in L_{k+1}^I(\varphi)$. The claim follows by induction. \square

Using the definition of I - k -models, the language of infinite sequences of environment inputs representable by lassos of length k can be expressed as $L_k^I(\Sigma^\omega)$.

Definition 2 (k-lasso-precise Implementations). *For a linear-time property φ over $\Sigma = 2^{AP}$, subset $I \subseteq AP$ of atomic propositions, and bound $k \in \mathbb{N}$, we say that a transition system \mathcal{T} is a k -lasso-precise implementation of φ , denoted $\mathcal{T} \models_{k,I} \varphi$, if it holds that $L_k^I(L(\mathcal{T})) \subseteq \varphi$.*

That is, in a k -lasso-precise implementation \mathcal{T} all the traces of \mathcal{T} that belong to the language $L_k^I(\Sigma^\omega)$ are I - k -models of the specification φ .

Problem definition: Synthesis of Lasso-Precise Implementations

Given a linear-time property φ over atomic propositions AP with input atomic propositions I , and given a bound $k \in \mathbb{N}$, construct an implementation \mathcal{T} such that $\mathcal{T} \models_{k,I} \varphi$, or determine that such an implementation does not exist.

Another way to bound the behaviour of the environment is to consider a bound on the size of its state space. The *synthesis problem for bounded environments* asks for a given linear temporal property φ and a bound $k \in \mathbb{N}$ to synthesize a transition system \mathcal{T} such that for every possible environment \mathcal{E} of size at most k , the transition system \mathcal{T} satisfies φ under environment \mathcal{E} , i.e., $\mathcal{T} \models_{\mathcal{E}} \varphi$.

We now establish the relationship between the synthesis of lasso-precise implementations and synthesis under bounded environments. Intuitively, the two synthesis problems can be reduced to each other since an environment of a given size, interacting with a given implementation, can only produce ultimately periodic sequences of inputs representable by lassos of length determined by the sizes of the environment and the implementation. This intuition is formalized in the following proposition, stating the connection between the two problems.

Proposition 1. *Given a specification φ over a set of atomic propositions AP with subset $I \subseteq AP$ of atomic propositions controlled by the environment, and a bound $k \in \mathbb{N}$, for every transition system \mathcal{T} the following statements hold:*

- (1) *If $\mathcal{T} \models_{\mathcal{E}} \varphi$ for all environments \mathcal{E} of size at most k , then $\mathcal{T} \models_{k,I} \varphi$.*
- (2) *If $\mathcal{T} \models_{k \cdot |\mathcal{T}|, I} \varphi$, then $\mathcal{T} \models_{\mathcal{E}} \varphi$ for all environments \mathcal{E} of size at most k .*

Proof. For (1), let \mathcal{T} be a transition system such that $\mathcal{T} \models_{\mathcal{E}} \varphi$ for all environments \mathcal{E} of size at most k . Assume, for the sake of contradiction, that $\mathcal{T} \not\models_{k,I} \varphi$. Thus, that there exists a word $\sigma \in L(\mathcal{T})$, such that $\sigma \in L_k^I(\Sigma^\omega)$ and $\sigma \not\models \varphi$.

Since $\sigma \in L_k^I(\Sigma^\omega)$, we can construct an environment \mathcal{E} of size at most k that produces the sequence of inputs $\sigma|_I$. Since \mathcal{E} is of size at most k , we have that $\mathcal{T} \models_{\mathcal{E}} \varphi$. Thus, since $\sigma \in L_{\mathcal{E}}(\mathcal{T})$, we have $\sigma \models \varphi$, which is a contradiction.

For (2), let \mathcal{T} be a transition system such that $\mathcal{T} \models_{k \cdot |\mathcal{T}|, I} \varphi$. Assume, for the sake of contradiction that there exists an environment \mathcal{E} of size at most k such that $\mathcal{T} \not\models_{\mathcal{E}} \varphi$. Since $\mathcal{T} \not\models_{\mathcal{E}} \varphi$, there exists $\sigma \in L_{\mathcal{E}}(\mathcal{T})$ such that $\sigma \not\models \varphi$. As the number of states of \mathcal{E} is at most k , the input sequences it generates can be represented as lassos of size $k \cdot |\mathcal{T}|$. Thus, $\sigma \in L_{k \cdot |\mathcal{T}|}^I(\Sigma^\omega)$. This is a contradiction with the choice of \mathcal{T} , according to which $\mathcal{T} \models_{k \cdot |\mathcal{T}|, I} \varphi$. \square

4.2 Automata-Theoretic Synthesis of Lasso-Precise Implementations

We now provide an automata-theoretic algorithm for the synthesis of lasso-precise implementations. The underlying idea of this approach is to first construct an automaton over finite traces that accepts all finite prefixes of traces in $L_k^I(\Sigma^\omega)$. Then, combining this automaton and an automaton representing the property φ we can construct an automaton whose language is non-empty if and only if there exists an k -lasso-precise implementation of φ .

The next theorem presents the construction of a deterministic finite automaton for the language $\text{Prefix}(L_k^I(\Sigma^\omega))$.

Theorem 2. *For any set AP of atomic propositions, subset $I \subseteq AP$, and bound $k \in \mathbb{N}$ there is a deterministic finite automaton \mathcal{A}_k over alphabet $\Sigma = 2^{AP}$, with size $(2^{|I|} + 1)^k \cdot (k + 1)^k$, such that $L(\mathcal{A}_k) = \{w \in \Sigma^* \mid \exists \sigma \in L_k^I(\Sigma^\omega). w < \sigma\}$.*

Idea & Construction. For given $k \in \mathbb{N}$ we first define an automaton $\widehat{\mathcal{A}}_k = (Q, q_0, \delta, F)$ over $\widehat{\Sigma} = 2^I$, such that $L(\widehat{\mathcal{A}}_k) = \{\widehat{w} \in \widehat{\Sigma}^* \mid \exists \widehat{\sigma} \in L_k^I(\widehat{\Sigma}^\omega). \widehat{w} < \widehat{\sigma}\}$. That, is $L(\widehat{\mathcal{A}}_k)$ is the set of all finite prefixes of infinite words over $\widehat{\Sigma}$ that can be represented by a lasso of length k . We can then define the automaton \mathcal{A}_k as the automaton that for each $w \in \Sigma^*$ simulates $\widehat{\mathcal{A}}_k$ on the projection $w|_I$ of w .

We define the automaton $\widehat{\mathcal{A}}_k = (Q, q_0, \delta, F)$ such that

- $Q = (\widehat{\Sigma} \cup \{\#\})^k \times \{-, 1, \dots, k\}^k$,
- $q_0 = (\#, (1, 2, \dots, k))$,

$$\delta(q, \alpha) = \begin{cases} (w \cdot \alpha \cdot \#^{m-1}, t) & \text{if } q = (w \cdot \#^m, t) \text{ where } 1 \leq m \leq k, \\ & w \in \widehat{\Sigma}^{(k-m)}, t \in \{-, 1, \dots, k\}^k \\ (w, (i'_1, \dots, i'_k)) & \text{if } q = (w, (i_1, \dots, i_k)) \text{ where } w \in \widehat{\Sigma}^k, \text{ and} \\ & i'_j = \begin{cases} - & i_j \leq k \wedge w(i_j) \neq \alpha \text{ or } i_j = - \\ i_j + 1 & i_j < k \wedge w(i_j) = \alpha \\ j & i_j = k \wedge w(i_j) = \alpha \end{cases} \\ - F = Q \setminus \{(w, (-, \dots, -)) \mid w \in \widehat{\Sigma}^k\}. \end{cases}$$

Proof. States of the form $(w \cdot \alpha \cdot \#^m, t)$ with $m \geq 1$ store the portion of the input word read so far, for input words of length smaller than k . In states of this form we have $t = (1, 2, \dots, k)$, which implies that all such states are accepting. In turn, this means that \mathcal{A}_k accepts all words of length smaller or equal to k . This is justified by the fact that, each word of length smaller or equal to k is a prefix of an infinite word in $L_k^I(\widehat{\Sigma}^\omega)$, obtained by repeating the prefix infinitely often. Now, let us consider words of length greater than k .

In states of the form $(u, (i_1, \dots, i_k))$, with $u \in \widehat{\Sigma}^*$, the word u stores the first k letters of the input word. Intuitively, the tuple (i_1, \dots, i_k) stores the information about the loops that are still possible, given the portion of the input word that is read thus far. To see this, let us consider a word $w \in \widehat{\Sigma}^*$ such that $|w| = l > k$, and let $q_0 q_1 \dots q_l$ be the run of \mathcal{A}_k on w . The state q_l is of the form $q_l = (w(1) \dots w(k), (i'_1, \dots, i'_k))$. It can be shown by induction on l that for each j we have $i'_j \neq -$ if and only if w is of the form $w = w' \cdot w'' \cdot w'''$ where $w' = w(1) \dots w(j-1)$, $w'' = (w(j) \dots w(k))^k$ for some $k \geq 0$, and $w''' = (w(j) \dots w(i'_j - 1))$. Thus, if $i'_j \neq -$, then it is possible to have a loop starting at position j , and i'_j is such that $(w(j) \dots w(i'_j - 1))$ is the prefix of $w(j) \dots w(k)$ appearing after the (possibly empty) sequence of repetitions of $w(j) \dots w(k)$. This means, that if $i'_j \neq -$, then w is a prefix of the infinite word $w' \cdot (w'')^\omega \in L_k^I(\widehat{\Sigma}^\omega)$. Therefore, if the run of \mathcal{A}_k on a word w with $|w| > k$ is accepting, then there exists $\sigma \in L_k^I(\widehat{\Sigma}^\omega)$ such that $w < \sigma$.

For the other direction, suppose that for each j , we have $i'_j = -$. Take any j , and consider the first position m in the run $q_0 q_1 \dots q_l$ where $i'_j = -$. By the definition of δ we have that $w(m) \neq w(i_j^{m-1})$. This means that the prefix $w(1) \dots w(m)$ cannot be extended to the word $w(1) \dots w(j-1)(w(j) \dots w(k))^\omega$. Since for every $j \in \{1, \dots, k\}$ we can find such a position m , it holds that there does not exist $\sigma \in L_k^I(\widehat{\Sigma}^\omega)$ such that $w < \sigma$. This concludes the proof. \square

The automaton constructed in the previous theorem has size which is exponential in the length of the lassos. In the next theorem we show that this exponential blow-up is unavoidable. That is, we show that every nondeterministic finite automaton for the language $\text{Prefix}(L_k^I(\Sigma^\omega))$ is of size at least $2^{\Omega(k)}$.

Theorem 3. For any bound $k \in \mathbb{N}$ and sets of atomic propositions AP and $\emptyset \neq I \subseteq AP$, every nondeterministic finite automaton \mathcal{N} over the alphabet $\Sigma = 2^{AP}$ that recognizes $L = \{w \in \Sigma^* \mid \exists \sigma \in L_k^I(\Sigma^\omega). w < \sigma\}$ is of size at least $2^{\Omega(k)}$.

Proof. Let $\mathcal{N} = (Q, Q_0, \delta, F)$ be a nondeterministic finite automaton for L . For each $w \in \Sigma^k$, we have that $w \cdot w \in L$. Therefore, for each $w \in \Sigma^k$ there exists at least one accepting run $\rho = q_0 q_1 \dots q_f$ of \mathcal{N} on $w \cdot w$. We denote with $q(\rho, m)$ the state q_m that appears at the position indexed m of a run ρ .

Let $a \in 2^I$ be a letter in 2^I , and let $\Sigma' = \Sigma \setminus \{a' \in \Sigma \mid a'|_I = a\}$. Let $L' \subseteq L$ be the language $L' = \{w \in \Sigma^k \mid \exists w' \in (\Sigma')^{k-1}, a' \in \Sigma : w = w' \cdot a' \text{ and } a'|_I = a\}$. That is, L' consists of the words of length k in which letters a' with $a'|_I = a$ appear in the last position and only in the last position.

Let us define the set of states

$$Q_k = \{q(\rho, k) \mid \exists w \in L' : \rho \text{ is an accepting run of } \mathcal{N} \text{ on } w \cdot w\}.$$

That is, Q_k consists of the states that appear at position k on some accepting run on some word $w \cdot w$, where w is from L' . We will show that $|Q_k| \geq 2^{k-1}$.

Assume that this does not hold, i.e., $|Q_k| < 2^{k-1}$. Since $|L'| \geq 2^{k-1}$, this implies that there exist $w_1, w_2 \in L'$, such that $w_1|_I \neq w_2|_I$ and there exists accepting runs ρ_1 and ρ_2 of \mathcal{N} on $w_1 \cdot w_1$ and $w_2 \cdot w_2$ respectively, such that $q(\rho_1, k) = q(\rho_2, k)$. That is, there must be two words in L' with $w_1|_I \neq w_2|_I$, which have accepting runs on $w_1 \cdot w_1$ and $w_2 \cdot w_2$ visiting the same state at position k .

We now construct a run $\rho_{1,2}$ on the word $w_1 \cdot w_2$ that follows ρ_1 for the first k steps on w_1 , ending in state $q(\rho_1, k)$, and from there on follows ρ_2 on w_2 . It is easy to see that $\rho_{1,2}$ is a run on the word $w_1 \cdot w_2$. The run is accepting, since ρ_2 is accepting. This means that $w_1 \cdot w_2 \in L$, which we will show leads to contradiction.

To see this, recall that $w_1 = w'_1 \cdot a'$ and $w_2 = w'_2 \cdot a''$, and $w_1|_I \neq w_2|_I$, and $a'|_I = a''|_I = a$. Since $w_1 \cdot w_2 \in L$, we have that $w'_1 \cdot a' \cdot w'_2 \cdot a'' < \sigma$ for some $\sigma \in L_k^I(\Sigma^\omega)$. That is, there exists a lasso for some word σ , and $w'_1 \cdot a' \cdot w'_2 \cdot a''$ is a prefix of this word. Since a does not appear in $w'_2|_I$, this means that the loop in this lasso is the whole word $w_1|_I$, which is not possible, since $w_1|_I \neq w_2|_I$.

This is a contradiction, which shows that $|Q| \geq |Q_k| \geq 2^{k-1}$. Since \mathcal{N} was an arbitrary nondeterministic finite automaton for L , this implies that the minimal automaton for L has at least $2^{\Omega(k)}$ states, which concludes the proof. \square

Using the automaton from Theorem 2, we can transform every property automaton \mathcal{A} into an automaton that accepts words representable by lassos of length less than or equal to k if and only if they are in $L(\mathcal{A})$, and accepts all words that are not representable by lassos of length less than or equal to k .

Theorem 4. Let AP be a set of atomic propositions, and let $I \subseteq AP$. For every (deterministic, nondeterministic or alternating) parity automaton \mathcal{A} over $\Sigma = 2^{AP}$, and $k \in \mathbb{N}$, there is a (deterministic, nondeterministic or alternating) parity automaton \mathcal{A}' of size $2^{O(k)} \cdot |\mathcal{A}|$, s.t., $L(\mathcal{A}') = (L_k^I(\Sigma^\omega) \cap L(\mathcal{A})) \cup (\Sigma^\omega \setminus L_k^I(\Sigma^\omega))$.

Proof. The theorem is a consequence of Theorem 2 established as follows. Let $\mathcal{A} = (Q, Q_0, \delta, \mu)$ be a parity automaton, and let $\mathcal{D} = (\widehat{Q}, \widehat{q}_0, \widehat{\delta}, F)$ be the deterministic finite automaton for bound k defined as in Theorem 2. We define the parity automaton $\mathcal{A}' = (Q', Q'_0, \delta', \mu')$ with the following components:

- $Q' = (Q \times \widehat{Q})$;
- $Q'_0 = \{(q_0, \widehat{q}_0) \mid q_0 \in Q_0\}$ (when \mathcal{A} is deterministic Q'_0 is a singleton set);
- $\delta'((q, \widehat{q}), \alpha) = \delta(q, \alpha)_{[q'/(q', \widehat{\delta}(\widehat{q}, \alpha))]}$, where $\delta(q, \alpha)_{[q'/(q', \widehat{\delta}(\widehat{q}, \alpha))]}$ is the Boolean expression obtained from $\delta(q, \alpha)$ by replacing every state q' by the state (q', \widehat{q}') ;
- $\mu'((q, \widehat{q})) = \begin{cases} \mu(q) & \text{if } \widehat{q} \in F, \\ 0 & \text{if } \widehat{q} \notin F. \end{cases}$

Intuitively, the automaton \mathcal{A}' is constructed as the product of \mathcal{A} and \mathcal{D} , where runs entering a state in \mathcal{D} that is not accepting in \mathcal{D} are accepting in \mathcal{A}' . To see this, recall from the construction in Theorem 2 that once \mathcal{D} enters a state in $\widehat{Q} \setminus \widehat{F}$ it remains in such a state forever. Thus, by setting the color of all states (q, \widehat{q}) where $\widehat{q} \notin F$ to 0, we ensure that words containing a prefix rejected by \mathcal{D} have only runs in which the highest color appearing infinitely often is 0. Thus, we ensure that all words that are not representable by lassos of length less than or equal to k are accepted by \mathcal{A}' , while words representable by lassos of length less than or equal to k are accepted if and only if they are in $L(\mathcal{A})$. \square

The following theorem is a consequence of the one above, and provides us with an automata-theoretic approach to solving the lasso-precise synthesis problem.

Theorem 5 (Synthesis). *Let AP be a set of atomic propositions, and $I \subseteq AP$ be a subset of AP consisting of the atomic propositions controlled by the environment. For a specification, given as a deterministic parity automaton \mathcal{P} over the alphabet $\Sigma = 2^{AP}$, and a bound $k \in \mathbb{N}$, finding an implementation \mathcal{T} , such that, $\mathcal{T} \models_{k,I} \mathcal{P}$ can be done in time polynomial in the size of the automaton \mathcal{P} and exponential in the bound k .*

5 Bounded Synthesis of Lasso-Precise Implementations

For a specification φ given as an LTL formula, a bound n on the size of the synthesized implementation and a bound k on the lassos of input sequences, *bounded synthesis of lasso-precise implementations* searches for an implementation \mathcal{T} of size n , such that $\mathcal{T} \models_{k,I} \varphi$. Using the automata constructions in the previous section we can construct a universal co-Büchi automaton for the language $L_k^I(\varphi) \cup (\Sigma^\omega \setminus L_k^I(\Sigma^\omega))$ and construct the constraint system as presented in [4]. This constraint system is exponential in both $|\varphi|$ and k . In the following we show how the problem can be encoded as a quantified Boolean formula of size polynomial in $|\varphi|$ and k .

Theorem 6. For a specification given as an LTL formula φ , and bounds $k \in \mathbb{N}$ and $n \in \mathbb{N}$, there exists a quantified Boolean formula ϕ , such that, ϕ is satisfiable if and only if there is a transition system $\mathcal{T} = (T, t_0, \tau, o)$ of size n with $\mathcal{T} \models_{k,I} \varphi$. The size of ϕ is in $O(|\varphi| + n^2 + k^2)$. The number of variables of ϕ is equal to $n \cdot (n \cdot 2^{|I|} + |O|) + k \cdot (|I| + 1) + n \cdot k(|O| + n + 1)$.

Construction. We encode the bounded synthesis problem in the following quantified Boolean formula:

$$\exists \{\tau_{t,i,t'} \mid t, t' \in T, i \in 2^I\}. \exists \{o_t \mid t \in T, o \in O\}. \quad (1)$$

$$\forall \{i_j \mid i \in I, 0 \leq j < k\}. \forall \{l_j \mid 0 \leq j < k\}. \quad (2)$$

$$\forall \{o_j \mid o \in O, 0 \leq j < n \cdot k\}. \quad (3)$$

$$\forall \{t_j \mid t \in T, 0 \leq j < n \cdot k\}. \quad (4)$$

$$\forall \{l'_j \mid 0 \leq j < n \cdot k\}. \quad (5)$$

$$\varphi_{\text{det}} \wedge (\varphi_{\text{lasso}} \wedge \varphi_{\in \mathcal{T}}^{n,k} \rightarrow [\![\varphi]\!]_0^{k,n \cdot k}) \quad (6)$$

which we read as: there is a transition system (1), such that, for all input sequences representable by lassos of length k (2) the corresponding sequence of outputs of the system (3) satisfies φ . The variables introduced in lines (4) and (5) are necessary to encode the corresponding output for the chosen input lasso.

An assignment to the variables satisfies the formula in line (6), if it represents a deterministic transition system (φ_{det}) in which lassos of length $n \cdot k$ ($\varphi_{\text{lasso}} \wedge \varphi_{\in \mathcal{T}}^{n,k}$) satisfy the property φ ($[\![\varphi]\!]_0^{(k,n \cdot k)}$). These constraints are defined as follows.

φ_{det} : A transition system is deterministic if for each state t and input i there is exactly one transition $\tau_{t,i,t'}$ to some state t' : $\bigwedge_{t \in T} \bigwedge_{i \in 2^I} \bigvee_{t' \in T} (\tau_{t,i,t'} \wedge \bigwedge_{t'' \neq t'} \overline{\tau_{t,i,t''}})$.

$\varphi_{\in \mathcal{T}}^{n,k}$: for a certain input lasso of size k we can match a lasso in the system of size at most $n \cdot k$. A lasso of this size in the transition system matches the input lasso if the following constraints are satisfied.

$$\bigwedge_{0 \leq j < n \cdot k} \bigwedge_{t \in T} (t_j \rightarrow \bigwedge_{o \in O} (o_j \leftrightarrow o_{t_j})) \quad (7)$$

$$\wedge \quad t_{00} \quad (8)$$

$$\wedge \quad \bigwedge_{0 \leq j < n \cdot k - 1} \bigwedge_{i \in 2^I} \bigwedge_{t,t' \in T} ((\bigwedge_{0 \leq j' < k} l_{j'} \rightarrow i_{\Delta(j,k,j')}) \wedge t_j \rightarrow (\tau_{t,i,t'} \leftrightarrow t'_{j+1})) \quad (9)$$

$$\wedge \quad \bigwedge_{i \in 2^I, t,t' \in T} ((\bigwedge_{0 \leq j' < k} l_{j'} \rightarrow i_{\Delta(n \cdot k - 1, k, j')}) \wedge t_{n \cdot k - 1} \rightarrow (\tau_{t,i,t'} \leftrightarrow (\bigvee_{0 \leq j < n \cdot k} l'_j \wedge t'_j))) \quad (10)$$

Lines (9) and (10) make sure that the chosen lasso follows the guessed transition relation τ . Line (10) handles the loop transition of the lasso, and makes sure that the loop of the lasso follows τ . Line (7) is a necessary requirement in order to match the output produced on the lasso with φ . If the output variables o_j satisfy the constraint $[\![\varphi]\!]_0^{(k,n \cdot k)}$, then the lasso satisfies φ . As the input lasso is

smaller than its matching lasso in the system we need to make sure that the indices of the input variables are correct with respect to the chosen loop. This is computed using the function Δ which is given by:

$$\Delta(j, k, j') = \begin{cases} j & \text{if } j < k, \\ ((j - k) \bmod (k - j')) + j' & \text{otherwise.} \end{cases}$$

φ_{lasso} : The formula encodes the additional constraint that exactly one of the loop variables can be true for a given variable valuation.

$\llbracket \varphi \rrbracket_0^{k,m}$: This constraint encodes the satisfaction of φ on lassos of size m . The encoding is similar to the encoding of bounded model checking [3], with the distinction of encoding the satisfaction relation of the atomic propositions, given below. As the inputs run with different indices than the outputs, we again, as in the lines (9) and (10), need to compute the correct indices using the function Δ .

	$h < m$	$h = m$
$\llbracket i \rrbracket_h^{k,m}$	$\bigwedge_{0 \leq j' < k} (l_{j'} \rightarrow i_{\Delta(h,k,j')})$	$\bigvee_{j=0}^{m-1} (l'_j \wedge \bigwedge_{0 \leq j' < k} (l_{j'} \rightarrow i_{\Delta(j,k,j')}))$
$\llbracket \neg i \rrbracket_h^{k,m}$	$\bigwedge_{0 \leq j' < k} (l_{j'} \rightarrow \neg i_{\Delta(h,k,j')})$	$\bigvee_{j=0}^{m-1} (l'_j \wedge \bigwedge_{0 \leq j' < k} (l_{j'} \rightarrow \neg i_{\Delta(j,k,j')}))$
$\llbracket o \rrbracket_h^{k,m}$	o_h	$\bigvee_{j=0}^{m-1} (l'_j \wedge o_j)$
$\llbracket \neg o \rrbracket_h^{k,m}$	$\neg o_h$	$\bigvee_{j=0}^{m-1} (l'_j \wedge \neg o_j)$

6 Synthesis of Approximate Implementations

In some cases, specifications remain unrealizable even when considered under bounded environments. Nevertheless, one might still be able to construct implementations that satisfy the specification in almost all input sequences of the environment. Consider for example the following simplified arbiter specification:

$$\square(\bar{w} \rightarrow \bigcirc \bar{g}) \wedge \square(r \rightarrow \lozenge g)$$

The specification defines an arbiter that should give grants g upon requests r , but is not allowed to provide these grants unless a signal w is true. The specification is unrealizable, because a sequence of inputs where the signal w is always false prevents the arbiter from answering any request. Bounding the environment does not help in this case as a lasso of size 1 already suffices to violate the specification (the one where w is always false). Nevertheless, one can still find reasonable implementations that satisfy the specification for a large fraction of input sequences. In particular, the fraction of input sequences where w remains false forever is less probable.

Definition 3 (ϵ -k-Approximation). For a specification φ , a bound k , and an error rate ϵ , we say that a transition system T approximately satisfies φ with an error rate ϵ for lassos of length at most k , denoted by $T \models_{k,I}^\epsilon \varphi$, if and only if, $\frac{|\{\sigma | \sigma \in L_k^I(L(T)), \sigma \models \varphi\}|}{|L_k^I((2^I)^\omega)|} \geq 1 - \epsilon$. We call T an ϵ - k -approximation of φ .

Theorem 7. For a specification given as a deterministic parity automaton P , a bound k and a error rate $0 \leq \epsilon \leq 1$, checking whether there is an implementation T , such that, $T \models_{k,I}^\epsilon P$ can be done in time polynomial in $|P|$ and exponential in k .

Proof. For a given ϵ and k , we construct a nondeterministic parity tree automaton \mathcal{N} that accepts all ϵ - k -approximations with respect to $L(P)$. For ϵ , we can compute the minimal number m of lassos from $L_k^I((2^I)^\omega)$ for which an ϵ - k -approximation has to satisfy the specification. In its initial state, the automaton \mathcal{N} guesses m many lassos and accepts a transition system if it does not violate the specification on any of these lassos. The latter check is done by following the structure of the automaton constructed for P using Theorem 4. In order to check whether there is an ϵ - k -approximation for P , we solve the emptiness game of \mathcal{N} . The size of \mathcal{N} is $(2^k)^{m+1} \cdot |P|$. \square

6.1 Symbolic Approach

In the following, we present a symbolic approach for finding ϵ - k -approximations based on maximum model counting. We show that we can build a constraint system and apply a maximum model counting algorithm to compute a transition system that satisfies a specification for a maximum number of input sequences.

Definition 4 (Maximum Model Counting [5]). Let X, Y and Z be sets of propositional variables and ϕ be a formula over X, Y and Z . Let x denote an assignment to X , y an assignment to Y , and z an assignment to Z . The maximum model counting problem for ϕ over X and Y is computing a solution for $\max_x \#y. \exists z. \phi(x, y, z)$.

For a specification φ , bounds k and n on the length of the lassos and size of the system, respectively, we can compute an ϵ - k -approximation for φ by applying a maximum model counting algorithm to the constraint system given below. It encodes transition systems of size n that have an input lasso of length k that satisfies φ .

$$\exists \{\tau_{t,i,t'} \mid t, t' \in T, i \in 2^I\}. \exists \{o_t \mid t \in T, o \in O\}. \quad (11)$$

$$\exists \{i_j \mid i \in I, 0 \leq j < k\}. \exists \{l_j \mid 0 \leq j < k\}. \quad (12)$$

$$\exists \{x_j^i \mid x \in I, 0 \leq i, j < k\} \quad (13)$$

$$\exists \{o_j \mid o \in O, 0 \leq j < n \cdot k\}. \quad (14)$$

$$\exists \{t_j \mid t \in T, 0 \leq j < n \cdot k\}. \quad (15)$$

$$\exists \{t'_j \mid 0 \leq j < n \cdot k\}. \quad (16)$$

$$\varphi_{\text{det}} \wedge \varphi_{\text{lasso}} \wedge \varphi_{\in T}^{n,k} \wedge \llbracket \varphi \rrbracket_0^{k,n \cdot k} \wedge \llbracket k \rrbracket_0 \quad (17)$$

To check the existence of a ϵ - k -approximation, we maximize over the set of assignment to variables that define the transition system (line 11) and count over variables that define input sequences of the environment given by lassos of length k . As two input lassos of the same length may induce the same infinite input sequence, we count over auxiliary variables that represent unrollings of the lassos instead of counting over the input propositions themselves (line 13).

The formulas φ_{det} , φ_{lasso} , $\varphi_{\in T}^{n,k}$ and $\llbracket \varphi \rrbracket_0^{k,n \cdot k}$ are defined as in the previous section. The formula $\llbracket k \rrbracket_0$ is defined over that variables in line (13) and makes sure that input lasso that represent the same infinite sequence are not counted twice by unrolling the lasso to size $2k$.

Theorem 8. *For a specification given as an LTL formula φ , and bounds k and n , and an error rate ϵ , the propositional formula ϕ defined above is of size $O(|\varphi| + n^2 + k^2)$. The number of variables of ϕ is equal to $n \cdot (n \cdot 2^{|I|} + |O|) + k \cdot (k \cdot |I| + |I| + 1) + n \cdot k(|O| + n + 1)$.*

7 Experimental Results

We implemented the symbolic encodings for the exact and approximate synthesis methods, and evaluated our approach on a bounded version of the greedy arbiter specification given in Sect. 1, and another specification of a round-robin arbiter. The round-robin arbiter is defined by the specification:

$$\square \diamond w \rightarrow \square \diamond g_1 \wedge \square \diamond g_2 \wedge \square(\neg w \rightarrow \bigcirc(\neg g_1 \wedge \neg g_2)) \wedge \square(\neg g_1 \vee \neg g_2)$$

This specification is realizable, with transition systems of size at least 4. We used our implementation to check whether we can find approximative solutions with smaller sizes. We used the tool CAQE [10] for solving the QBF instances and the tool MaxCount [5] for solving the approximate synthesis instances.

Table 1. Experimental results for the symbolic approaches. The rate in the approximate approach is the rate of input lassos on which the specification is satisfied.

Instance	QBF							MaxCount					
	Spec.	Proc.	#States	Bound	Result	#Gates	\forall	\exists	Time	#Max	#Count	Rate	Time
Round-Robin Arbiter	2	2	4	Unreal	15556	48	12	9.91 s		12	8	0.5	26 s
	2	3	2	Unreal	5338	40	24	2.45 s		24	4	0.88	161 s
	2	4	2	Real	13414	60	12	12.15 s		40	4	0.88	283 s
Greedy Arbiter	1	2	2	Real	1597	20	10	0.41 s		10	4	1.0	0.79 s
	1	2	3	Unreal	4749	30	10	1.95 s		10	6	0.88	3.86 s
	1	3	3	Unreal	16861	48	21	17.26 s		21	6	0.88	20.83 s
	1	4	3	Real	43692	78	36	3 min 7.44 s	36	6	1.0	2 min 43 s	
	1	4	4	-	169829	104	36	TO		36	8	-	TO
	2	4	2	Real	24688	62	72	1 min. 24 s	72	6	-	-	TO
	2	4	3	Unreal	103433	93	72	27 min 15.2	72	12	-	-	TO
	3	2	2	Unreal	3985	93	72	1.39 s		38	8	0.65	4.18 s

The results are presented in Table 1. As usual in synthesis, the size of the instances grows quickly as the size bound and number of processes increase. Inspecting the encoding constraints shows that the constraint for the specification is responsible for more than 80% of the number of gates in the encoding. The results show that, using the approach we proposed, we can synthesize implementations for unrealizable specifications by bounding the environment. The results for the approximate synthesis method further demonstrate that for the unrealizable cases one can still obtain approximative implementations that satisfy the specification on a large number of input sequences.

8 Conclusion

In many cases, the unrealizability of a specification is due to the assumption that the environment has unlimited power in producing inputs to the system. In this paper, we have investigated the problem of synthesizing implementations under bounded environment behaviors. We have presented algorithms for solving the synthesis problem for bounded lassos and the synthesis of approximate implementations that satisfy the specification up to a certain rate.

We have also provided polynomial encodings of the problems into quantified Boolean formulas and maximum model counting instances. Our experiments demonstrate the principal feasibility of the approach. Our experiments also show that the instances can quickly become large. While this is a common phenomenon for synthesis, there clearly is a lot of room for optimization and experimentation with both the solvers for quantified Boolean expressions and for maximum model counting.

References

1. Alur, R., Moarref, S., Topcu, U.: Counter-strategy guided refinement of GR(1) temporal logic specifications. In: Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20–23, 2013, pp. 26–33. IEEE (2013)
2. Chatterjee, K., Henzinger, T.A., Jobstmann, B.: Environment assumptions for synthesis. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 147–161. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85361-9_14
3. Clarke, E., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. *Form. Methods Syst. Des.* **19**(1), 7–34 (2001)
4. Finkbeiner, B., Schewe, S.: Bounded synthesis. *Int. J. Software Tools Technol. Transf.* **15**(5–6), 519–539 (2013)
5. Fremont, D.J., Rabe, M.N., Seshia, S.A.: Maximum model counting. Technical Report UCB/EECS-2016-169, EECS Department, University of California, Berkeley, Nov 2016. This is the extended version of a paper to appear at AAAI 2017
6. Kupferman, O., Lustig, Y., Vardi, M.Y., Yannakakis, M.: Temporal synthesis for bounded systems and environments. In: Schwentick, T., Dürr, C. (eds.) 28th International Symposium on Theoretical Aspects of Computer Science, STACS 2011, March 10–12, 2011, Dortmund, Germany, vol. 9 of LIPIcs, pages 615–626. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2011)

7. Kwiatkowska, M., Parker, D.: Automated verification and strategy synthesis for probabilistic systems. In: Van Hung, D., Ogawa, M. (eds.) ATVA 2013. LNCS, vol. 8172, pp. 5–22. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-02444-8_2
8. Li, W., Dworkin, L., Seshia, S.A.: Mining assumptions for synthesis. In: Singh, S., Jobstmann, B., Kishinevsky, M., Brandt, J. (eds.) 9th IEEE/ACM International Conference on Formal Methods and Models for Codesign, MEMOCODE 2011, Cambridge, UK, 11–13 July, 2011, pp. 43–50. IEEE (2011)
9. Pnueli, A.: The temporal logic of programs. In: Proceedings of the 18th Annual Symposium on Foundations of Computer Science, SFCS 1977, Washington, DC, USA, 1977. IEEE Computer Society (1977)
10. Rabe, M.N., Tentrup, L.: Caqe: a certifying QBF solver. In: Proceedings of the 15th Conference on Formal Methods in Computer-aided Design (FMCAD 2015), pp. 136–143, September 2015
11. Raman, V., Lignos, C., Finucane, C., Lee, K.C.T., Marcus, M.P., Kress-Gazit, H.: Sorry dave, i'm afraid I can't do that: explaining unachievable robot tasks using natural language. In: Newman, P., Fox, D., Hsu, D. (eds.), Robotics: Science and Systems IX, Technische Universität Berlin, Berlin, Germany, June 24 - June 28, 2013 (2013)
12. Rozier, K.Y.: Specification: the biggest bottleneck in formal methods and autonomy. In: Blazy, S., Chechik, M. (eds.) VSTTE 2016. LNCS, vol. 9971, pp. 8–26. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48869-1_2
13. Vardi, M.Y.: Nontraditional applications of automata theory. In: Hagiya, M., Mitchell, J.C. (eds.) TACS 1994. LNCS, vol. 789, pp. 575–597. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-57887-0_116
14. Vardi, M.Y.: Alternating automata and program verification. In: van Leeuwen, J. (ed.) Computer Science Today. LNCS, vol. 1000, pp. 471–485. Springer, Heidelberg (1995). <https://doi.org/10.1007/BFb0015261>

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Quantified Invariants via Syntax-Guided Synthesis

Grigory Fedyukovich¹(✉) Sumanth Prabhu²,
Kumar Madhukar², and Aarti Gupta¹

¹ Princeton University, Princeton, USA

{grigoryf,aartig}@cs.princeton.edu

² TCS Research, Pune, India

{sumanth.prabhu,kumar.madhukar}@tcs.com



Abstract. Programs with arrays are ubiquitous. Automated reasoning about arrays necessitates discovering properties about ranges of elements at certain program points. Such properties are formally specified by universally quantified formulas, which are difficult to find, and difficult to prove inductive. In this paper, we propose an algorithm based on an enumerative search that discovers quantified invariants in stages. First, by exploiting the program syntax, it identifies ranges of elements accessed in each loop. Second, it identifies potentially useful facts about individual elements and generalizes them to hypotheses about entire ranges. Finally, by applying recent advances of SMT solving, the algorithm filters out wrong hypotheses. The combination of properties is often enough to prove that the program meets a safety specification. The algorithm has been implemented in a solver for Constrained Horn Clauses, FREQHORN, and extended to deal with multiple (possibly nested) loops. We show that FREQHORN advances state-of-the-art on a wide range of public array-handling programs.

1 Introduction

Formally verifying programs against safety specifications is difficult. This problem worsens in the presence of data structures like lists, arrays, and maps, which are ubiquitous in real-world applications. For instance, proving an array-handling program safe often requires discovering an inductive invariant that is universally quantified over ranges of array elements. Such invariants help to prove the unreachability of error states independently of the size of the array. However, the majority of invariant synthesis approaches are limited to quantifier-free numerical invariants. The approach presented in this paper advances the knowledge by an effective technique to discover quantified invariants over arrays and linear integer arithmetic.

Syntax-guided techniques [3] have recently been applied to synthesize quantifier-free numerical invariants [15–17,34] in the approach called FREQHORN. In a nutshell, FREQHORN collects various statistics from the syntactical patterns occurring in the program’s source code and uses them to construct a

© The Author(s) 2019

I. Dillig and S. Tasiran (Eds.): CAV 2019, LNCS 11561, pp. 259–277, 2019.

https://doi.org/10.1007/978-3-030-25540-4_14

set of formal grammars that specify a search space for invariants. It is often sufficient to perform an *enumerative search* over the formulas produced from these grammars and identify a set of suitable inductive invariants among them using an off-the-shelf solver for Satisfiability Modulo Theories (SMT). The presence of arrays complicates this reasoning in a few respects: it is hard to find suitable candidates and difficult to prove them inductive.

In this paper, we present a novel technique that extends the approach of enumerative search in general, and its instantiation in FREQHORN in particular, to reason about quantifiers. It discovers invariants over arrays in multiple stages. First, by exploiting the program syntax, it identifies ranges of elements accessed in each loop. Second, it identifies potentially useful facts about individual elements and generalizes them to hypotheses about entire ranges. The SMT-based validation of candidates, which are quantified formulas, is often inexpensive as they are constructed using the same syntactic patterns that appear in the source code. Furthermore, for supporting certain corner cases, our approach allows specifying additional rules that help in generalizing learned properties. The combination of properties proven inductive by an SMT solver is often enough to prove that the program meets a safety specification.

We show that FREQHORN advances state-of-the-art on a selection of array-handling programs from SVCOMP¹ and literature. For instance, it can prove completely automatically that an array is monotone after applying a sorting algorithm. Furthermore, FREQHORN is able to discover quantifier-free invariants over integer variables in the program, use them as inductive relatives while checking inductiveness of quantified candidates over arrays; and vice versa.

While a detailed discussion of the related work comes later in the paper (Sect. 6), it is noteworthy that being syntax-guided crucially helps us overcome several limitations of other techniques to verify array-handling programs [2, 9, 11, 35]. Most of them avoid inferring quantified invariants explicitly and thus do not produce checkable proofs. As a result, tools are fragile and in practice often output false positives (see Sect. 5 for concrete results). By comparison, our approach never produces false positives, and its results can be validated by existing SMT solvers.

The core contributions made through this work are:

- a novel syntax-guided approach to generate universally quantified invariants for programs manipulating arrays;
- an algorithm and its fully automated implementation; and
- a thorough experimental evaluation comparing our technique with state-of-the-art in verification of array-handling programs.

The rest of the paper is structured as follows. In Sect. 2, we give background and notation and illustrate our approach on an example. Our main contributions are then presented in Sect. 3 (main algorithm) and Sect. 4 (important design choices). In Sect. 5, we show the evaluation and comparison with state-of-the-art. Finally, the related work and conclusion complete the paper in Sects. 6 and 7, respectively.

¹ Software Verification Competition, <http://sv-comp.sosy-lab.org/>.

2 Background

The Satisfiability Modulo Theories (SMT) task is to decide whether there is an assignment m of values to variables in a first-order logic formula φ that makes it true. We write $\varphi \implies \psi$, if every satisfying assignment to φ is also a satisfying assignment to some formula ψ . By $Expr$ we denote the space of all possible quantifier-free formulas in our background theory and by $Vars$ a range of possible variables.

2.1 Programs as Constrained Horn Clauses

To guarantee expected behaviors, programs require proofs, such as inductive invariants, ranking functions, or recurrence sets. It is becoming increasingly popular to consider a verification task as a *proof synthesis* task which is formulated as a system of SMT formulas involving unknown predicates, also known as *constrained Horn clauses* (CHC). The synthesis goal is to discover a suitable interpretation of all unknown predicates that make all CHCs true. CHCs offer the advantages of flexibility and modularity in designing verifiers for various systems and languages. CHCs can be constructed in a way that captures the operational semantics of a language in question, and an off-the-shelf CHC solver can be used for solving the resulting formulas.

Definition 1. A linear constrained Horn clause (*CHC*) over a set of uninterpreted relation symbols \mathcal{R} is a formula in first-order logic that has the form of one of three implications (called respectively a fact, an inductive clause, and a query):

$$\begin{aligned} \varphi(\vec{x}_1) &\implies \mathbf{inv}_1(\vec{x}_1) \\ \mathbf{inv}_1(\vec{x}_1) \wedge \varphi(\vec{x}_1, \vec{x}_2) &\implies \mathbf{inv}_2(\vec{x}_2) \\ \mathbf{inv}_1(\vec{x}_1) \wedge \varphi(\vec{x}_1) &\implies \perp \end{aligned}$$

where $\mathbf{inv}_1, \mathbf{inv}_2 \in \mathcal{R}$ are uninterpreted symbols, \vec{x}_1, \vec{x}_2 are vectors of variables, and φ , called a body, is a fully interpreted formula (i.e., φ does not have applications of \mathbf{inv}_1 or \mathbf{inv}_2).

For a CHC C , by $src(C)$ we denote an application of $\mathbf{inv} \in \mathcal{R}$ in the premise of C (if C is a fact, we write $src(C) \stackrel{\text{def}}{=} \top$). Similarly, by $dst(C)$ we denote an application of $\mathbf{inv} \in \mathcal{R}$ in the conclusion of C (if C is a query, we write $dst(C) \stackrel{\text{def}}{=} \perp$). We define functions rel and $args$, such that for each $\mathbf{inv}(\vec{x})$, $rel(\mathbf{inv}(\vec{x})) \stackrel{\text{def}}{=} \mathbf{inv}$ and $args(\mathbf{inv}(\vec{x})) \stackrel{\text{def}}{=} \vec{x}$. For a CHC C , by $body(C)$ we denote the body (i.e., φ) of C .

Example 1. Figure 1 gives a program in the C programming language that handles two integer arrays, \mathbf{A} and \mathbf{B} , both of an unknown size N . The \mathbf{A} array has unknown content, and the program first identifies a value m which is smaller or

```

int N = nondetInt();
int *A = nondetArray(N);
int m = 0;
for (int i = N - 1; i ≥ 0; i--) { if (m > A[i]) m = A[i]; }
int *B = malloc(N*sizeof(int));
for (int i = 0; i < N; i++) { B[N - i - 1] = A[i] - m; }
int s = 0;
for (int i = 0; i < N; i++) { s = s + B[i]; }
assert(s ≥ 0);

```

Fig. 1. Example program: source code in C.

- (A) $i' = N' - 1 \wedge m' = 0 \implies \mathbf{inv}_1(A', i', m', N')$
- (B) $\mathbf{inv}_1(A, i, m, N) \wedge i \geq 0 \wedge m' = \text{ite}(m > A[i], A[i], m) \wedge i' = i - 1 \implies \mathbf{inv}_1(A, i', m', N)$
- (C) $\mathbf{inv}_1(A, i, m, N) \wedge i < 0 \wedge i' = 0 \implies \mathbf{inv}_2(A, B, i', m, N)$
- (D) $\mathbf{inv}_2(A, B, i, m, N) \wedge i < N \wedge B' = \text{store}(B, N - i - 1, A[i] - m) \wedge i' = i + 1 \implies \mathbf{inv}_2(A, B', i', m, N)$
- (E) $\mathbf{inv}_2(A, B, i, m, N) \wedge i \geq N \wedge i' = 0 \wedge s' = 0 \implies \mathbf{inv}_3(A, B, i', m, s', N)$
- (F) $\mathbf{inv}_3(A, B, i, m, s, N) \wedge i < N \wedge s' = s + B[i] \wedge i' = i + 1 \implies \mathbf{inv}_3(A, B, i', m, s', N)$
- (G) $\mathbf{inv}_3(A, B, i, m, s, N) \wedge i \geq N \wedge s < 0 \implies \perp$

Fig. 2. Example program: CHC encoding.

equal to all elements of A (it might be either a minimal element among the content of A or 0). Then, the program populates B by values of A with m subtracted. Interestingly, the order of elements A and B is not preserved, e.g., $A[0] - m$ gets written to $B[N - 1]$, and so on. Finally, the program computes the sum s of all elements in B and requires us to prove that s is never negative.

Figure 2 gives a CHC encoding of the program. The system has three uninterpreted predicates, \mathbf{inv}_1 , \mathbf{inv}_2 , and \mathbf{inv}_3 corresponding to invariants at heads of the three loops. The primed variables correspond to modified variables. Rules **B**, **D**, and **F** encode the loop bodies, and the remaining rules encode the fragments of code before, after, or between the loops. In particular, rule **G** ensures that after the third loop has terminated, a program state with a negative value of s is unreachable. Before we describe how our technique solves this CHC system (see Sect. 2.2), we briefly introduce the notion of satisfiability of CHCs.

Definition 2. Given a set of uninterpreted relation symbols \mathcal{R} and a set S of CHCs over \mathcal{R} , we say that S is satisfiable if there exists an interpretation that assigns to each n -ary symbol $\mathbf{inv} \in \mathcal{R}$ a relation over n -tuples and makes all implications in S valid.

In the paper, we assume that a relation assigned by an interpretation is represented by a formula ψ over at most n free variables.

We call a CHC C inductive when $\text{rel}(\text{src}(C)) = \text{rel}(\text{dst}(C)) = \mathbf{inv}$ for some \mathbf{inv} . While accessing an array in a loop, we assume the existence of an integer counter variable. More formally:

Definition 3. Let C be an inductive CHC, $\vec{x} = \text{args}(\text{src}(C))$, and $\vec{x}' = \text{args}(\text{dst}(C))$. We say that C is array-handling if there exist numbers c and a , such that (1) $1 \leq c \leq |\vec{x}|$ and $1 \leq a \leq |\vec{x}'|$; (2) $\vec{x}[c]$ (and consequently, its “primed copy” $\vec{x}'[c]$) has type integer, (3) either of these implications holds:

$$\text{body}(C) \implies \vec{x}[c] < \vec{x}'[c] \quad (1)$$

$$\text{body}(C) \implies \vec{x}[c] > \vec{x}'[c] \quad (2)$$

(4) $\vec{x}[a]$ (and consequently $\vec{x}'[a]$) has type array, and (5) there is an access function f that identifies a relationship between an access to $\vec{x}[a]$ in $\text{body}(C)$ and $\vec{x}'[c]$.

2.2 Illustrating Example

The CHC system in Fig. 2 has a solution, indicating that the program meets its specification. In particular:

$$\begin{aligned} \mathbf{inv}_1 \mapsto & \forall j . i < j < N \implies m \leq A[j] \\ \mathbf{inv}_2 \mapsto & \forall j . 0 \leq j < N \implies m \leq A[j] \wedge \\ & \forall j . 0 \leq j < i \implies B[N - j - 1] = A[j] - m \\ \mathbf{inv}_3 \mapsto & \forall j . 0 \leq j < N \implies m \leq A[j] \wedge \\ & \forall j . 0 \leq j < N \implies B[N - j - 1] = A[j] - m \\ & \wedge s \geq 0 \end{aligned}$$

The interpretation of \mathbf{inv}_1 means that as the first loop progresses (i.e., all elements $A[N - 1], A[N - 2], \dots, A[i + 1]$ are sequentially considered), the value of m is always smaller than all the considered elements. Thus, we refer to the interpretation of \mathbf{inv}_1 as a *progress lemma*. When the first loop has terminated, clearly, this property holds for all elements from $A[0]$ to $A[N - 1]$. Because A leaks through the second loop without any changes, the interpretation of \mathbf{inv}_1 gets finalized (thus, it becomes a *finalized lemma*) and added to an interpretation of \mathbf{inv}_2 .

Additionally, the interpretation of \mathbf{inv}_2 gets a relational fact about pairs of elements $A[0]$ and $B[N - 1], A[1]$ and $B[N - 2], \dots, A[i - 1]$ and $B[N - i - 2]$, which again appears as a progress lemma and then gets finalized in an interpretation of \mathbf{inv}_3 . With these two quantified invariants about all elements of A , and relation about pairs of elements of A and B , it is possible to derive the remaining lemma in the interpretation of \mathbf{inv}_3 , namely, $s \geq 0$; which concludes the proof.

3 Invariants via Enumerative Search

In this work, we aim at discovering a solution for a CHC system S over a set of uninterpreted symbols \mathcal{R} enumeratively, i.e., by guessing a candidate formula for each $\mathbf{inv} \in \mathcal{R}$, substituting it for all CHCs $C \in S$ and checking their validity.

3.1 Quantifier-Free Invariants

We build on top of an algorithm, called FREQHORN, recently proposed in [17]. Its key insight is an automatic construction of a set of formal grammars $G(\mathbf{inv})$ for each $\mathbf{inv} \in \mathcal{R}$ based on either source code, program behaviors, or both. Importantly, these grammars are *conjunction-free*: they cannot be used to produce a conjunction of clauses and can give rise to only a finite number of formulas, potentially related to invariants (otherwise, the approach does not guarantee strong convergence). Since invariants are often represented by a conjunction of lemmas, FREQHORN attempts to sample (i.e., recursively apply production rules) each lemma from a grammar in separation, until a combination of them is sufficient for the inductiveness and safety, or a search space is exhausted. FREQHORN relies on an SMT solver to filter out unsuccessfully sampled lemmas.

The construction of formal grammars is biased by the syntax of CHC encoding. First, FREQHORN collects a set of *Seeds* by converting the body of each CHC to a Conjunctive Normal Form, extracting, and normalizing each conjunct. Then, the set of seeds could be optionally replenished by a set of *behavioral seeds* and *bounded proofs*. They are constructed respectively from the concrete values of variables obtained from actual program runs, and Craig interpolants from unsatisfiable finite unrollings of the CHC systems. Finally, the production rules are created in a way to enable producing seeds and also their *mutants* (i.e., syntactically similar formulas to seeds). In general, no specific restriction on a grammar-construction method is imposed; so in practice, the grammars are allowed to be more (or less) general to enable a broader (or more focused) search space for invariants.

3.2 Quantified Candidates from Quantifier-Free Grammars

The main obstacle for applying the enumerative search to generate array invariants is that the grammars do not allow quantifiers. Because grammars are constructed automatically from syntactic patterns which appear in the original programs, in the presence of arrays, we can expect expressions involving only particular elements of arrays (such as ones accessed via a loop counter). However, since each loop repeats certain operations over a *range* of array elements, we have to *generalize* the extracted expressions about individual elements to expressions about entire ranges.

Let a set of variables associated with a relation symbol \mathbf{inv} be $Vars(\mathbf{inv}) \stackrel{\text{def}}{=} IntVars(\mathbf{inv}) \cup ArrVars(\mathbf{inv})$, where $IntVars(\mathbf{inv})$ and $ArrVars(\mathbf{inv})$ are disjoint and contain integer variables and array variables, respectively. A candidate quantified invariant over arrays consists of three parts:

- a set of quantified integer variables $QVars(\mathbf{inv})$, which are introduced by our algorithm and do not appear in $Vars(\mathbf{inv})$;
- a *range* formula over $QVars(\mathbf{inv}) \cup IntVars(\mathbf{inv})$; and
- a quantifier-free *cell property* over $QVars(\mathbf{inv}) \cup Vars(\mathbf{inv})$.

Algorithm 1. PREPARE(S, \mathcal{R})

Input: CHCs S over \mathcal{R}
Output: Formal grammars $G(\mathbf{inv})$, quantified variables $QVars(\mathbf{inv})$ and $progressRange(\mathbf{inv})$ for each $\mathbf{inv} \in \mathcal{R}$

```

1 for each  $\mathbf{inv} \in \mathcal{R}$  do
2    $Seeds \leftarrow SYNTSEEDS(\mathbf{inv}) \cup BEHAVSEEDS(\mathbf{inv});$ 
3    $cnt \leftarrow GETCOUNTERS(S, \mathbf{inv}, ArrVars(\mathbf{inv}));$ 
4   if  $\emptyset \neq cnt$  then
5      $QVars(\mathbf{inv}) \leftarrow COPY(cnt);$ 
6      $progressRange(\mathbf{inv}) \leftarrow GETRANGE(cnt);$ 
7    $G(\mathbf{inv}) \leftarrow REPLACE(GETGRAMMAR(Seeds), cnt, QVars(\mathbf{inv}));$ 

```

Algorithm 2. SOLVEARRAYCHCs(S, \mathcal{R})

Input: CHCs S over \mathcal{R}
Output: $res \in \{\text{SAT}, \text{UNKNOWN}\}$, $Lemmas : \mathcal{R} \rightarrow 2^{Expr}$

```

1  $\langle G, QVars, progressRange \rangle \leftarrow \text{PREPARE}(S, \mathcal{R});$ 
2 for each  $\mathbf{inv} \in \mathcal{R}$  do  $Lemmas(\mathbf{inv}) \leftarrow \emptyset;$ 
3 while  $\exists C \in S . \left( \bigwedge_{\ell \in Lemmas(\text{rel}(src(C)))} \ell(args(src(C))) \wedge body(C) \Rightarrow \perp \right)$  do
4   if  $\forall \mathbf{inv} \in \mathcal{R} . \text{ALLBLOCKED}(G(\mathbf{inv}))$  then return  $\langle \text{UNKNOWN}, \emptyset \rangle;$ 
5    $\mathbf{inv} \leftarrow \text{PICKLOOP}(\mathcal{R});$ 
6   if  $QVars(\mathbf{inv}) = \emptyset$  then  $Cand(\mathbf{inv}) \leftarrow \text{SAMPLE}(G(\mathbf{inv}));$ 
7   else  $Cand(\mathbf{inv}) \leftarrow \forall QVars(\mathbf{inv}).$ 
       $QVars(\mathbf{inv}) \in progressRange(\mathbf{inv}) \implies \text{SAMPLE}(G(\mathbf{inv}));$ 
8    $ExtCand \leftarrow \text{EXTEND}(S, \{\mathbf{inv}\}, Cand, Lemmas);$ 
9   if  $\forall \mathbf{inv}' \in \mathcal{R} . ExtCand(\mathbf{inv}') = \top$  then  $G(\mathbf{inv}) \leftarrow \text{BLOCK}(G, Cand, \mathbf{inv});$ 
10  else
11    for each  $\mathbf{inv}' \in \mathcal{R}$  do
12       $Lemmas(\mathbf{inv}') \leftarrow Lemmas(\mathbf{inv}') \cup \{ExtCand(\mathbf{inv}')\};$ 
13       $G(\mathbf{inv}') \leftarrow \text{BLOCK}(G, ExtCand, \mathbf{inv}');$ 
14 return  $\langle \text{SAT}, Lemmas \rangle;$ 

```

A naive idea for getting a range formula and a cell property is to sample them separately, and then to bind them together using some $QVars(\mathbf{inv})$. But it would result in a large search space. Algorithm 1 gives a more tailored procedure on the matter. The central role in this process is taken by an analysis of the loop counters which are used to access array elements (line 3). This analysis is performed once for each loop before the main verification process, and thus its results are reused in all iterations of the verification process.

Our algorithm identifies $QVars(\mathbf{inv})$ by creating a fresh variable for each counter, including counters of nested loops (line 5). It then generates range formulas based on the results of the analysis (line 6) such that: (1) the range formula itself is an inductive invariant for \mathbf{inv} , and (2) the range formula is expressed over the initial values of counters of \mathbf{inv} and the counters themselves. Finally, only a cell property is going to be produced from the grammar $G(\mathbf{inv})$,

Algorithm 3. WEAKEN(S' , \mathcal{R}' , $Cand$, $Lemmas$)

Input: CHCs S' over \mathcal{R}' , candidates $Cand(\mathbf{inv})$; learned $Lemmas(\mathbf{inv})$ for each $\mathbf{inv} \in \mathcal{R}'$

Output: weakened $Cand$

```

1 if  $toRecheck \leftarrow \perp$ ;
2 for all  $C \in S'$  do
3   if  $\bigwedge_{\ell \in Lemmas(rel(src(C)))} \ell(args(src(C))) \wedge Cand(rel(src(C)))(args(src(C))) \wedge$ 
     $body(C) \not\Rightarrow Cand(rel(dst(C)))(args(dst(C)))$  then
4     if ISFINALIZEDARRAYCAND( $Cand, rel(dst(C))$ ) then
5        $Cand(rel(dst(C))) \leftarrow GETREGRESSCAND(Cand, rel(dst(C)))$ ;
6     else
7        $Cand(rel(dst(C))) \leftarrow \top$ ;
8      $toRecheck \leftarrow \top$ ;
9     break;
10 if  $toRecheck$  then return WEAKEN( $S'$ ,  $\mathcal{R}'$ ,  $Cand$ ,  $Lemmas$ );
11 else return  $Cand$ ;
```

Algorithm 4. EXTEND(S , \mathcal{R} , $Cand$, $Lemmas$), cf [17].

Input: CHCs S over \mathcal{R} ; $\mathcal{R}' \subseteq \mathcal{R}$, candidates $Cand(\mathbf{inv})$; learned $Lemmas(\mathbf{inv})$ for each $\mathbf{inv} \in \mathcal{R}'$

Output: extended $Cand$

```

1  $Cand \leftarrow WEAKEN(S', \mathcal{R}', Cand, Lemmas)$ ;
2 for all  $C \in S$  s.t.  $rel(src(C)) \in \mathcal{R}'$  do
3    $Cand(rel(dst(C))) \leftarrow PROPAGATE(C, Cand)$ ;
4    $Cand \leftarrow EXTEND(S, \mathcal{R}' \cup \{rel(dst(C))\}, Cand, Lemmas)$ ;
5 return  $Cand$ ;
```

constructed from the seeds (recall Sect. 3.1), in which all counters are replaced by the corresponding variables from $QVars(\mathbf{inv})$ (line 7). Thus, the only part of the candidate formula where the counter can appear is the range formula.

Once grammars, $QVars$, and ranges are detected, our approach proceeds to sample candidates and to check them with an SMT solver. The general flow of this algorithm is illustrated in Algorithm 2. For each $\mathbf{inv} \in \mathcal{R}$, it initiates a set $Lemmas(\mathbf{inv})$ (line 2). Then it iteratively guesses lemmas until a combination of them is inductive and safe, or a search space is exhausted (lines 3–4).

Compared to the baseline approach from [17], our new algorithm fixes a shape for the candidates for arrays. At the same time, it permits to sample quantifier-free candidates (line 6): they could be either formulas over counters or any other variables in the loop, or even formulas over isolated array elements (if, e.g., accessed by a constant). Then (line 8), Algorithm 2 propagates candidates through all available implications in CHCs using quantifier elimination and identifies lemmas among the candidates. This step is similar to the baseline

approach from [17], but for completeness of presentation, we provide the pseudocode in Algorithms 3 and 4. The only differences are (1) in the implementation of the candidate propagation for array candidates and (2) in the weakening of failed candidates (both in Algorithm 3, to be discussed in Sects. 4.3 and 4.4, respectively).

Both successful and unsuccessful candidates are “blocked” from their grammars to avoid re-sampling them in the next iterations. This fact together with the property of grammars being conjunction-free gives the main hint for proving the following theorem.

Theorem 1. *Algorithm 2 always makes a finite number of iterations, and if it returns with SAT then the CHC system is satisfiable.*

Next section discusses a particular instantiation of important subroutines that make our invariant synthesizer effective in practice.

4 Design Choices

Our main contribution is a completely automated algorithm for finding quantified invariants for array-handling loops. In this section, we first show how by exploiting the program syntax we can identify ranges of elements accessed in each loop (Sect. 4.1). Second, we present an intuitive justification to why our candidates can often be proved as lemmas by an off-the-shelf SMT solver (Sect. 4.2). Finally, we extend our algorithm to handle more complicated cases of multiple loops (Sects. 4.3–4.4), and benchmarks of the tiling [9] technique, which are adapted from the industrial code of battery controllers (Sect. 4.5).

4.1 Discovery of Progress Lemmas

We start with the simplest scenario of a single loop handling just one array. Let S be a system of CHCs over a set of uninterpreted relation symbols \mathcal{R} . Let $\mathbf{inv} \in \mathcal{R}$ correspond to a loop, in which arrays are accessed using some counter variable i (counters are automatically identified by posing and solving queries of forms (1) and (2)).

Recall that we do not necessarily require the array elements to be accessed directly by i , and we allow an access function f to identify relationships between i and an index of the accessed element. However, we assume that the counter is unique in the loop because it is the case in most of the practical applications. In principle, our algorithm can be extended to loops handling several independent counters (although it is rare in practice), with the help of additionally discovered lemmas that describe relationships among counters. We leave a discussion about this to future work.

Definition 4. *A range of \mathbf{inv} and a counter i is a formula over $\text{IntVars}(\mathbf{inv})$ and a free variable v having form $L < v \wedge v < U$, such that either of formulas $L < i$ or $i < U$ is a lemma for \mathbf{inv} . A progress lemma is either a formula $L < v \wedge v < i$ (if $L < i$ is a lemma), or a formula $i < v \wedge v < U$ (if $i < U$ is a lemma).*

Both ranges and progress ranges can be identified statically. Let C_1 and C_2 be two CHCs, such that $\mathbf{inv} = \text{rel}(\text{dst}(C_1)) = \text{rel}(\text{src}(C_2)) = \text{rel}(\text{dst}(C_2))$ and $\mathbf{inv} \neq \text{rel}(\text{src}(C_1))$. It is common in practice that $\text{body}(C_1)$ identifies a symbolic bound b on the initial value of i : it could be either a lower bound (if i increments in $\text{body}(C_2)$) or an upper bound (if i decrements). In this case, a progress range of \mathbf{inv} is simply computed as a lemma for \mathbf{inv} over i and b . A range of \mathbf{inv} can often be constructed as a conjunction of the progress range with the negation of the termination condition of $\text{body}(C_2)$.²

Example 2. For the CHC-encoding of the program is shown in Fig. 2, the ranges of \mathbf{inv}_1 , \mathbf{inv}_2 and \mathbf{inv}_3 are all equal to $-1 < v < N$. The progress range of \mathbf{inv}_1 is $i < v < N$, and the progress ranges of \mathbf{inv}_2 and \mathbf{inv}_3 are $-1 < v < i$.

We call candidates, that use progress ranges in their left sides, *progress candidates*:

$$\forall \vec{q}. \text{progressRange}(\mathbf{inv})(\vec{q}) \implies \text{cand}$$

where $\vec{q} = Q\text{Vars}(\mathbf{inv})$ and cand is a quantifier-free formula over $Q\text{Vars}(\mathbf{inv}) \cup \text{Int Vars}(\mathbf{inv})$. As can be seen from Algorithm 1, all sampled candidates are progress candidates. However, during the next steps of the algorithm (i.e., propagation and weakening) we will use other kind of candidates (namely, *regress* and *finalized*, see Sects. 4.3 and 4.4 respectively).

If a progress candidate is proven inductive, we call it a *progress lemma*.

4.2 SMT-Based Inductiveness Checking

We rely on recent advances of SMT solving to identify successful candidates, a conjunction of which is directly used to prove the desired safety specification. In general, solving quantified formulas for validity is a hard task, however, in certain cases, the initiation and inductiveness queries can be simplified and reduced to a sequence of (sometimes even quantifier-free) formulas over integer arithmetic. We illustrate such proving strategy, inspired by the *tiling* approach [9], on the following example.

Example 3. Recall the CHC system from Fig. 2. Consider a progress candidate $\forall j. i < j < N \implies m \leq A[j]$ for \mathbf{inv}_1 . Checking its initiation (i.e., for CHC **A**) requires deciding validity of the following quantified formula:

$$i' = N' - 1 \wedge m' = 0 \implies \left(\forall j. i' < j < N' \implies m' \leq A'[j] \right) \quad (3)$$

The range formula $i' < j < N'$ simplifies to $N' - 1 < j < N'$, which is always false, making formula (3) always valid.

² Thus, we explicitly require guards of loops to have the forms of an inequality, which is the most common array access pattern.

Checking the inductiveness of the candidate (i.e., for CHC **B**) boils down to solving a more complicated formula:

$$\begin{aligned} & \left(\forall j . i < j < N \implies m \leq A[j] \right) \\ & \wedge i \geq 0 \wedge m' = \text{ite}(m > A[i], A[i], m) \wedge i' = i - 1 \implies \\ & \quad \left(\forall j . i' < j < N \implies m' \leq A[j] \right) \end{aligned} \quad (4)$$

Although quantifiers are present on both sides of (4), proving its validity is not hard. Indeed, the query is reducible to two implications:

$$\begin{aligned} & \left(\forall j . i < j < N \implies m \leq A[j] \right) \wedge m' = \text{ite}(m > A[i], A[i], m) \implies m' \leq A[i] \\ & \left(\forall j . i < j < N \implies m \leq A[j] \right) \wedge \\ & \quad m' = \text{ite}(m > A[i], A[i], m) \implies \left(\forall j . i < j < N \implies m' \leq A[j] \right) \end{aligned}$$

The former does not require any information about $A[i+1], \dots, A[N-1]$, so the entire quantified conjunction is ignored, and $A[i]$ could be replaced by a fresh integer variable. The latter is trickier: it requires to prove that if all elements in a range are greater or equal than m , then they are also greater or equal to $\text{ite}(m > A[i], A[i], m)$. This again is reduced to a quantifier-free formula over integer arithmetic:

$$m \leq A[j] \wedge m' = \text{ite}(m > A[i], A[i], m) \implies m' \leq A[j]$$

Thus, because formulas (3) and (4) are valid, the progress candidate is proved a progress lemma.

In general, we cannot always conduct proofs that easily. Often, the prerequisite for success is the commonality of an access function f in the candidate and the body of the CHC. Fortunately, our algorithm ensures that all access functions used in the candidates are borrowed directly from bodies of CHCs. Thus, in many cases, FREQHORN is able to check large amounts of candidates quickly.

4.3 Strategy of Lemma Propagation

In this subsection, we identify a useful strategy for propagation of quantified lemmas through adjacent CHCs in the given system, inspired by [17]. Let some $\mathbf{inv}_1 \in \mathcal{R}$ have the following lemma:

$$\forall \vec{q} . \rho(\vec{q}) \implies \ell$$

where $\vec{q} = Q\text{Vars}(\mathbf{inv}_1)$, formula ρ over $\vec{q} \cup \text{IntVars}(\mathbf{inv}_1)$ is either a range or a progress range, and ℓ is over $\vec{q} \cup \text{Vars}(\mathbf{inv}_1)$. Let then a CHC C be such that $\text{rel}(\text{src}(C)) = \mathbf{inv}_1$ and $\text{rel}(\text{dst}(C)) = \mathbf{inv}_2$, and its body be $\varphi(\vec{x}_1, \vec{x}_2)$.

Definition 5. Forward propagation of lemma $\forall \vec{q}. \rho(\vec{q}) \implies \ell$ through C gives a formula of the following form:

$$\forall \vec{q}. (\exists \vec{x}_1. \rho(\vec{q})(\vec{x}_1) \wedge \varphi(\vec{x}_1, \vec{x}_2)) \implies (\exists \vec{x}_1(\vec{x}_1, \vec{q}). \ell \wedge \varphi(\vec{x}_1, \vec{x}_2))$$

Example 4. Recall the example from Fig. 2 and the following lemma for \mathbf{inv}_1 :

$$\forall j. i < j < N \implies m \leq A[j]$$

The body of \mathbf{C} is $i < 0 \wedge i' = 0$, thus the forward propagation gives the following formula:

$$\forall j. (\exists i. i < j < N \wedge i < 0 \wedge i' = 0) \implies (\exists i. m \leq A[j] \wedge i < 0 \wedge i' = 0)$$

Applying quantifier elimination to both sides of the implication, we get the following formula:

$$\forall j. 0 \leq j < N \implies m \leq A[j]$$

Note that this formula is not going to be immediately learned as a lemma, but instead should be checked by the solver for inductiveness. Intuitively, such a candidate represents some facts about array elements that were accessed during a loop that has terminated. If after the propagation it appeared that the candidate uses the entire range then we refer to such candidate to as a *finalized* candidate.

4.4 Weakening Strategy

Whenever a finalized candidate cannot be proven inductive, we often do not want to withdraw it completely. Instead, our algorithm runs *weakening* and proposes *regress candidates*. The main idea is to calculate a range of elements which have not been touched by the loop yet. This is an inverse of the procedure outlined in Sect. 4.1.

Definition 6. Given $\mathbf{inv} \in \mathcal{R}$, its $\text{Range}(\mathbf{inv})$ and $\text{progressRange}(\mathbf{inv})$ formulas, we call a regress range a formula of the following kind:

$$\text{regressRange}(\mathbf{inv}) \stackrel{\text{def}}{=} \text{Range}(\mathbf{inv}) \wedge \neg \text{progressRange}(\mathbf{inv})$$

We call candidates that use regress ranges in their left sides as *regress candidates*. Clearly, a regress candidate is weaker than the corresponding finalized candidate. Thus, from the failure to prove inductiveness of the finalized candidate it does not follow that the regress candidate is not inductive; and it makes sense to try proving it in the next iteration.

4.5 Learning from Sub-ranges

In complicated scenarios of loops with multiple iterators, multiple array variables or multiple access functions, the iterative process of lemma discovery, might end up in a large number of quantified formulas and get lost while checking a candidate for inductiveness (recall Sect. 4.2). To overcome current limitations in existing SMT solvers, it appeared to be useful to help the solver while generalizing learned lemmas. In particular, a property could be learned for two subranges of an array, and then combined in the following way:

```

int N = nondetInt();
int *A = nondetArray(2*N);
int val1 = 1, val2 = 3, m = nondetInt();
for (int i = 1; i ≤ N; i++) {
    if (m < val2) A[2*i-2] = val2; else A[2*i-2] = 0;
    if (m < val1) A[2*i-1] = val1; else A[2*i-1] = 0; }
for (int i = 0; i < 2*N; i++) assert(A[i]==0 || A[i] ≤ m);

```

Fig. 3. Learning from sub-ranges.

Lemma 1. Let for some $\mathbf{inv} \in \mathcal{R}$ two lemmas be of the following kind:

$$\forall \vec{q}. \rho_1(\vec{q}) \implies \ell \quad \forall \vec{q}. \rho_2(\vec{q}) \implies \ell \quad (5)$$

Then, the following is also a lemma for \mathbf{inv} :

$$\forall \vec{q}. \rho_1(\vec{q}) \vee \rho_2(\vec{q}) \implies \ell$$

Example 5. Figure 3 shows a program from the tiling benchmark suite [9]. If lemmas $\forall j. 0 < j < N \implies A[2*j - 1] = 0 \vee A[2*j - 1] \leq m$ and $\forall j. 0 < j < N \implies A[2*j - 2] = 0 \vee A[2*j - 2] \leq m$ are discovered, then formula $\forall j. 0 \leq j < 2*N - 1 \implies A[j] = 0 \vee A[j] \leq m$ is also a lemma.

5 Evaluation

We have implemented our algorithm on top of the FREQHORN³ tool. It takes a system of CHCs with arrays as input and performs an enumerative search as presented in Sect. 4. The tool uses Z3 [12] to solve SMT queries.

We have evaluated FREQHORN on 137 satisfiable CHC-translations of publicly available C programs (whose assertions are safe) taken from the SVCOMP ReachSafety Array subcategory and literature. These programs include variations of standard array copying, initializing, maximum, minimum, sorting, and tiling benchmarks. Among these 137 benchmarks, 79 have a single loop, and 58 have multiple loops, including 7 that have nested loops. These programs are encoded using the theories of Arrays, Linear (LIA) and Non-linear Integer Arithmetic (NIA). Our experiments have been performed on an Ubuntu 18.04 machine running at 2.5 GHz and having 16 GB memory, with a timeout of 100 s for every benchmark. FREQHORN solved 129 benchmarks within the timeout, of which 73 solved benchmarks had a single loop and 56 had multiple loops.

We have compared our tool with SPACER (Z3 v4.8.3) [26], that implements a recent QUIC3 [22] algorithm, BOOSTER (v0.2) [2], VIAP (v1.0) [35], and VERI-ABS (v1.3.10) [11]. The last two tools performed well in the ReachSafety Array

³ The source code and benchmarks are available at <https://github.com/grigoryfedyukovich/aeval/tree/rnd>.

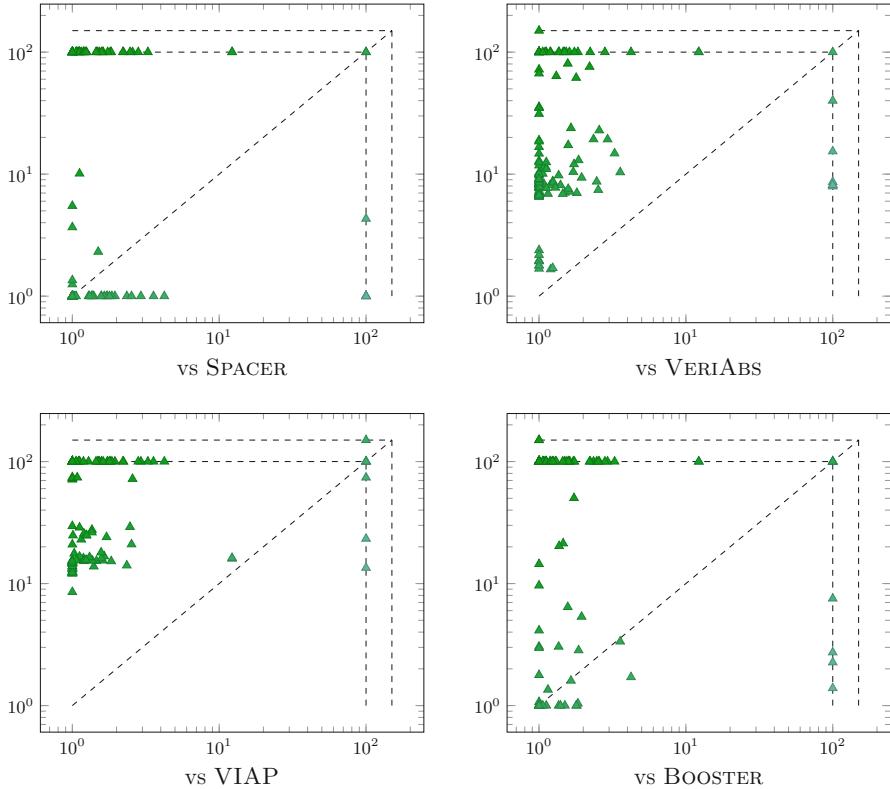


Fig. 4. FREQHORN vs competitors. Each point in a plot represents a pair of the run times (sec \times sec) of FREQHORN (x-axis) and a competitor (y-axis). Timeouts are placed on the inner dashed lines; false alarms, unsupported cases, and crashes are on the outer dashed lines.

subcategory at SVCOMP 2019⁴. Figure 4 gives a comparison of FREQHORN timings against timings of these tools.⁵

Compare to 129 benchmarks solved by FREQHORN, only 81 were solved by SPACER, 108 – by VERIABS, 70 – by VIAP, and 48 – by BOOSTER.

FREQHORN solved 54 benchmarks on which SPACER diverged. Our intuition is that SPACER works poorly on programs with non-deterministic assignments and NIA operations, which our tool can handle.

FREQHORN solved 27 benchmarks on which VERIABS diverged. VERIABS failed to solve programs with nested loops and when array values were dependent on access indices. Furthermore, it decided one of the programs as unsafe. Time-wise, FREQHORN significantly outperformed VERIABS on all benchmarks.

⁴ <https://sv-comp.sosy-lab.org/2019/results/results-verified/>.

⁵ The time taken for every benchmark is available at: <http://bit.ly/2VS5Mtf>.

Importantly, the short time taken by FREQHORN includes the time for generating a checkable witness – quantified invariant – an essence that VERIABS cannot produce by design. On the other side, VERIABS solved several benchmarks after merging loops. No quantified invariant satisfying the FREQHORN’s restrictions exists for these benchmarks before this program transformation.

FREQHORN solved 60 programs on which VIAP diverged. VIAP decided one program as unsafe. There were no programs on which FREQHORN took more time than VIAP. Finally, FREQHORN solved 83 programs on which BOOSTER diverged. And again, BOOSTER decided two programs as unsafe.

6 Related Work

Our algorithm for quantified invariant synthesis extends the prior work on checking satisfiability of CHCs [15–17], where solutions do not permit quantifiers. It works in a similar – enumerate-and-check – manner, but there are two crucial changes: (1) introduction of quantifiers, to formulate hypotheses over a subset of array indices, and (2) a generalization mechanism, to derive properties that may hold over the entire range of array indices.

Many existing approaches for verifying programs over arrays are extensions of well-known techniques for programs over scalar variables to quantified invariants. For example, by extending predicates with Skolem variables in predicate abstraction [30], by exploiting the MCMT [19] framework in lazy abstraction with interpolants [1] and its integration with acceleration [2], and, recently, QUIC3 [22], that extends IC3 [8, 14] to universally quantified invariants. Apart from the skeletal similarity, however, these approaches rely on orthogonal techniques.

Partitioning of arrays has also been used to infer invariants in many different ways. It refers to splitting an array into symbolic segments, and may be based on syntax [20, 23, 25] or semantics [10, 31]. Invariants may be inferred for each segment separately and generalized for the entire array. The partitioning need not be explicit, as in [13]. However, most of these techniques (except [13, 31]) are restricted to contiguous array segments, and work well when different loop iterations write to disjoint array locations or when the segments are non-overlapping. Tiling [9], a property-driven verification technique, overcomes these limitations for a class of programs by inferring array access patterns in loops. But identifying tiles of array accesses is itself a difficult problem, and the approach is currently based on heuristics developed by observing interesting patterns.

There are a number of approaches that verify array programs without inferring quantified invariants explicitly. A straightforward way is to smash all array elements into a single memory location [4], but it is quite imprecise. Every array element might also be considered a separate variable, but it is not possible with unknown array sizes. There are also techniques that abstract an array to a fixed number of elements, e.g. k -distinguished cell abstraction [32, 33] and k -shrinkability [24, 29]. Such abstractions usually reduce array modifying loops with unknown bounds to a known, small bound. It may even be possible to get rid of such loops altogether, by accelerating (computing transitive closures of)

transition relations involving array updates in that loop [7]. Along similar lines, VIAP [35] resorts to reasoning with recurrences instead of loops. It translates the input program, including loops, to a set of first-order axioms, and checks if they derive the property. But all these techniques do not obtain quantified invariants explicitly, unlike ours. Besides, many of these transformations produce an abstraction of the original program, i.e., they do not preserve safety.

Alternatively, there are approaches that use sufficiently expressive templates to infer quantified invariants over arrays [5, 21, 27]. However, the templates need to be supplied manually. For instance, [6] uses a template space of quantified invariants and reduces the problem to quantifier-free invariant generation. Thus, universally quantified solutions for unknown predicates in a CHC system may be obtained by extending a generic CHC solver to handle quantified predicates. Learning need not be limited to user-supplied templates; one may do away with the templates entirely and learn only from examples and counterexamples [18]. Alternatively, [36] chooses a template upfront and refurbishes it with constants or coefficients appearing in the program source. Similarly, [28] proposes to infer array invariants without any user guidance or any user-defined templates or predicates. Their method is based on automatic analysis of predicates that update an array and allows one to generate first-order invariants, including those that contain alternations of quantifiers. But it does not work for nested loops. By comparison, our technique supports multiple as well as nested loops, enables candidate propagation between loops and, more importantly, generates the grammar automatically from the syntactical constructions appearing in the program’s source.

7 Conclusion

We have presented a new algorithm to synthesize quantified invariants over array variables, systematically accessed in loops. Our algorithm implements an enumerative search that guesses invariants based on syntactic constructions which appear in the code and checks their initiation, inductiveness, and safety with an off-the-shelf SMT solver. Key insights behind our approach are that individual accesses to array elements performed in the loop can be generalized to hypotheses about entire ranges, and the existing SMT solvers can be used to validate these hypotheses efficiently. Our implementation on top of a CHC solver FREQHORN confirmed that such strategy is effective on a variety of practical examples. In a vast majority of cases, our tool outperformed competitors and provided checkable guarantees that prevented from reporting false positives.

Acknowledgements. This work was supported in part by NSF Grant 1525936. Any opinions, findings, and conclusions expressed herein are those of the authors and do not necessarily reflect those of the NSF.

References

1. Alberti, F., Bruttiomesso, R., Ghilardi, S., Ranise, S., Sharygina, N.: Lazy abstraction with interpolants for arrays. In: Bjørner, N., Voronkov, A. (eds.) LPAR 2012. LNCS, vol. 7180, pp. 46–61. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28717-6_7
2. Alberti, F., Ghilardi, S., Sharygina, N.: Booster: an acceleration-based verification framework for array programs. In: Cassez, F., Raskin, J.-F. (eds.) ATVA 2014. LNCS, vol. 8837, pp. 18–23. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11936-6_2
3. Alur, R., et al.: Syntax-guided synthesis. In: FMCAD, pp. 1–17. IEEE (2013)
4. Bertrane, J., et al.: Static analysis and verification of aerospace software by abstract interpretation. Found. Trends Program. Lang. **2**(2–3), 71–190 (2015)
5. Beyer, D., Henzinger, T.A., Majumdar, R., Rybalchenko, A.: Invariant Synthesis for combined theories. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 378–394. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-69738-1_27
6. Bjørner, N., McMillan, K., Rybalchenko, A.: On solving universally quantified horn clauses. In: Logozzo, F., Fähndrich, M. (eds.) SAS 2013. LNCS, vol. 7935, pp. 105–125. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38856-9_8
7. Bozga, M., Habermehl, P., Iosif, R., Konečný, F., Vojnar, T.: Automatic verification of integer array programs. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 157–172. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_15
8. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 70–87. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18275-4_7
9. Chakraborty, S., Gupta, A., Unadkat, D.: Verifying array manipulating programs by tiling. In: Ranzato, F. (ed.) SAS 2017. LNCS, vol. 10422, pp. 428–449. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66706-5_21
10. Cousot, P., Cousot, R., Logozzo, F.: A parametric segmentation functor for fully automatic and scalable array content analysis. In: POPL, pp. 105–118 (2011)
11. Darke, P., et al.: VeriAbs: verification by abstraction and test generation. In: Beyer, D., Huisman, M. (eds.) TACAS 2018, Part I. LNCS, vol. 10806, pp. 457–462. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89963-3_32
12. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
13. Dillig, I., Dillig, T., Aiken, A.: Fluid updates: beyond strong vs. weak updates. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 246–266. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11957-6_14
14. Eén, N., Mishchenko, A., Brayton, R.K.: Efficient implementation of property directed reachability. In: FMCAD, pp. 125–134. IEEE (2011)
15. Fedyukovich, G., Bodík, R.: Accelerating syntax-guided invariant synthesis. In: Beyer, D., Huisman, M. (eds.) TACAS 2018, Part I. LNCS, vol. 10805, pp. 251–269. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89960-2_14
16. Fedyukovich, G., Kaufman, S., Bodík, R.: Sampling invariants from frequency distributions. In: FMCAD, pp. 100–107. IEEE (2017)
17. Fedyukovich, G., Prabhu, S., Madhukar, K., Gupta, A.: Solving constrained horn clauses using syntax and data. In: FMCAD, pp. 170–178. IEEE (2018)

18. Garg, P., Löding, C., Madhusudan, P., Neider, D.: Learning universally quantified invariants of linear data structures. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 813–829. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_57
19. Ghilardi, S., Ranise, S.: MCMT: a model checker modulo theories. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS (LNAI), vol. 6173, pp. 22–29. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14203-1_3
20. Gopan, D., Reps, T., Sagiv, M.: A framework for numeric analysis of array operations. In: POPL, pp. 338–350 (2005)
21. Gulwani, S., McCloskey, B., Tiwari, A.: Lifting abstract interpreters to quantified logical domains. In: POPL, pp. 235–246. ACM (2008)
22. Gurfinkel, A., Shoham, S., Vizel, Y.: Quantifiers on demand. In: Lahiri, S.K., Wang, C. (eds.) ATVA 2018. LNCS, vol. 11138, pp. 248–266. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-01090-4_15
23. Halbwachs, N., Péron, M.: Discovering properties about arrays in simple programs. In: PLDI, pp. 339–348 (2008)
24. Jana, A., Khedker, U.P., Datar, A., Venkatesh, R., Niyas, C.: Scaling bounded model checking by transforming programs with arrays. In: Hermenegildo, M.V., Lopez-Garcia, P. (eds.) LOPSTR 2016. LNCS, vol. 10184, pp. 275–292. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63139-4_16
25. Jhala, R., McMillan, K.L.: Array abstractions from proofs. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 193–206. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73368-3_23
26. Komuravelli, A., Gurfinkel, A., Chaki, S.: SMT-based model checking for recursive programs. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 17–34. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_2
27. Kong, S., Jung, Y., David, C., Wang, B.-Y., Yi, K.: Automatically inferring quantified loop invariants by algorithmic learning from simple templates. In: Ueda, K. (ed.) APLAS 2010. LNCS, vol. 6461, pp. 328–343. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17164-2_23
28. Kovács, L., Voronkov, A.: Finding loop invariants for programs over arrays using a theorem prover. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 470–485. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00593-0_33
29. Kumar, S., Sanyal, A., Venkatesh, R., Shah, P.: Property checking array programs using loop shrinking. In: Beyer, D., Huisman, M. (eds.) TACAS 2018, Part I. LNCS, vol. 10805, pp. 213–231. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89960-2_12
30. Lahiri, S.K., Bryant, R.E.: Constructing quantified invariants via predicate abstraction. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 267–281. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24622-0_22
31. Liu, J., Rival, X.: Abstraction of arrays based on non contiguous partitions. In: D’Souza, D., Lal, A., Larsen, K.G. (eds.) VMCAI 2015. LNCS, vol. 8931, pp. 282–299. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46081-8_16
32. Monniaux, D., Alberti, F.: A simple abstraction of arrays and maps by program translation. In: Blazy, S., Jensen, T. (eds.) SAS 2015. LNCS, vol. 9291, pp. 217–234. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48288-9_13
33. Monniaux, D., Gonnord, L.: Cell morphing: from array programs to array-free horn clauses. In: Rival, X. (ed.) SAS 2016. LNCS, vol. 9837, pp. 361–382. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53413-7_18

34. Prabhu, S., Madhukar, K., Venkatesh, R.: Efficiently learning safety proofs from appearance as well as behaviours. In: Podelski, A. (ed.) SAS 2018. LNCS, vol. 11002, pp. 326–343. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99725-4_20
35. Rajkhowa, P., Lin, F.: Extending VIAP to handle array programs. In: Piskac, R., Rümmer, P. (eds.) VSTTE 2018. LNCS, vol. 11294, pp. 38–49. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03592-1_3
36. Sharma, R., Aiken, A.: From invariant checking to invariant inference using randomized search. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 88–105. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_6

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Efficient Synthesis with Probabilistic Constraints

Samuel Drews^(✉), Aws Albarghouthi, and Loris D’Antoni

University of Wisconsin-Madison, Madison, USA
`sedrews@wisc.edu`

Abstract. We consider the problem of synthesizing a program given a probabilistic specification of its desired behavior. Specifically, we study the recent paradigm of *distribution-guided inductive synthesis* (DIGITS), which iteratively calls a synthesizer on finite sample sets from a given distribution. We make theoretical and algorithmic contributions: (*i*) We prove the surprising result that DIGITS only requires a polynomial number of synthesizer calls in the size of the sample set, despite its ostensibly exponential behavior. (*ii*) We present a property-directed version of DIGITS that further reduces the number of synthesizer calls, drastically improving synthesis performance on a range of benchmarks.

1 Introduction

Over the past few years, progress in automatic program synthesis has touched many application domains, including automating data wrangling and data extraction tasks [2, 13, 15, 21, 22, 30], generating network configurations that meet user intents [10, 29], optimizing low-level code [25, 28], and more [4, 14].

The majority of the current work has focused on synthesis under Boolean constraints. However, often times we require the program to adhere to a probabilistic specification, e.g., a controller that succeeds with a high probability, a decision-making model operating over a probabilistic population model, a randomized algorithm ensuring privacy, etc. In this work, we are interested in (1) investigating probabilistic synthesis from a theoretical perspective and (2) developing efficient algorithmic techniques to tackle this problem.

Our starting point is our recent framework for probabilistic synthesis called *distribution-guided inductive synthesis* (DIGITS) [1]. The DIGITS framework is analogous in nature to the *guess-and-check* loop popularized by counterexample-guided approaches to synthesis and verification (CEGIS and CEGAR). The key idea of the algorithm is reducing the probabilistic synthesis problem to a non-probabilistic one that can be solved using existing techniques, e.g., SAT solvers. This is performed using the following loop: (1) approximating the input probability distribution with a finite sample set; (2) synthesizing a program for various possible output assignments of the finite sample set; and (3) invoking a probabilistic verifier to check if one of the synthesized programs indeed adheres to the given specification.

DIGITS has been shown to theoretically converge to correct programs when they exist—thanks to learning-theory guarantees. The primary bottleneck of DIGITS is the number of expensive calls to the synthesizer, which is ostensibly exponential in the size of the sample set. Motivated by this observation, this paper makes theoretical, algorithmic, and practical contributions:

- On the theoretical side, we present a detailed analysis of DIGITS and prove that it only requires a polynomial number of invocations of the synthesizer, explaining that the strong empirical performance of the algorithm is not merely due to the heuristics presented in [1] (Sect. 3).
- On the algorithmic side, we develop an improved version of DIGITS that is property-directed, in that it only invokes the synthesizer on instances that have a chance of resulting in a correct program, without sacrificing convergence. We call the new approach τ -DIGITS (Sect. 4).
- On the practical side, we implement τ -DIGITS for sketch-based synthesis and demonstrate its ability to converge significantly faster than DIGITS. We apply our technique to a range of benchmarks, including illustrative examples that elucidate our theoretical analysis, probabilistic repair problems of unfair programs, and probabilistic synthesis of controllers (Sect. 5).

2 An Overview of DIGITS

In this section, we present the synthesis problem, the DIGITS [1] algorithm, and fundamental background on learning theory.

2.1 Probabilistic Synthesis Problem

Program Model. As discussed in [1], DIGITS searches through some (infinite) set of programs, but it requires that the set of programs has *finite VC dimension* (we restate this condition in Sect. 2.3). Here we describe one constructive way of obtaining such sets of programs with finite VC dimension: we will consider sets of programs defined as *program sketches* [27] in the simple grammar from [1], where a program is written in a loop-free language, and “holes” defining the sketch replace some constant terminals in expressions.¹ The syntax of the language is defined below:

$$P := V \leftarrow E \mid \text{if } B \text{ then } P \text{ else } P \mid P P \mid \text{return } V$$

Here, P is a program, V is the set of variables appearing in P , E (resp. B) is the set of linear arithmetic (resp. Boolean) expressions over V (where, again, constants in E and B can be replaced with holes), and $V \leftarrow E$ is an assignment. We assume a vector v_I of variables in V that are inputs to the program. We

¹ In the case of loop-free program sketches as considered in our program model, we can convert the input-output relation into a real arithmetic formula that guaranteedly has finite VC dimension [12].

also assume there is a single Boolean variable $v_r \in V$ that is returned by the program.² All variables are real-valued or Boolean. Given a vector of constant values \mathbf{c} , where $|\mathbf{c}| = |\mathbf{v}_I|$, we use $P(\mathbf{c})$ to denote the result of executing P on the input \mathbf{c} .

In our setting, the inputs to a program are distributed according to some *joint probability distribution* \mathbb{D} over the variables \mathbf{v}_I . Semantically, a program P is denoted by a *distribution transformer* $\llbracket P \rrbracket$, whose input is a distribution over values of \mathbf{v}_I and whose output is a distribution over \mathbf{v}_I and v_r .

A program also has a *probabilistic postcondition*, $post$, defined as an inequality over terms of the form $\Pr[B]$, where B is a Boolean expression over \mathbf{v}_I and v_r . Specifically, a probabilistic postcondition consists of Boolean combinations of the form $e > c$, where $c \in \mathbb{R}$ and e is an arithmetic expression over terms of the form $\Pr[B]$, e.g., $\Pr[B_1]/\Pr[B_2] > 0.75$.

Given a triple $(P, \mathbb{D}, post)$, we say that P is *correct* with respect to \mathbb{D} and $post$, denoted $\llbracket P \rrbracket(\mathbb{D}) \models post$, iff $post$ is true on the distribution $\llbracket P \rrbracket(\mathbb{D})$.

Example 1. Consider the set of intervals of the form $[0, a] \subseteq [0, 1]$ and inputs x uniformly distributed over $[0, 1]$ (i.e. $\mathbb{D} = \text{Uniform}[0, 1]$). We can write inclusion in the interval as a (C-style) program (left) and consider a postcondition stating that the interval must include at least half the input probability mass (right):

```
if(0 <= x && x <= a) {
    return 1;
}
return 0;
```

$$\Pr_{x \sim \mathbb{D}}[P(x) = 1] \geq 0.5$$

Let P_c denote the interval program where a is replaced by a constant $c \in [0, 1]$. Observe that $\llbracket P_c \rrbracket(\mathbb{D})$ describes a joint distribution over (x, v_r) pairs, where $[0, c] \times \{1\}$ is assigned probability measure c and $(c, 1] \times \{0\}$ is assigned probability measure $1 - c$. Therefore, $\llbracket P_c \rrbracket(\mathbb{D}) \models post$ if and only if $c \in [0.5, 1]$.

Synthesis Problem. DIGITS outputs a program that is approximately “similar” to a given functional specification and that meets a postcondition. This functional specification is some input-output relation which we quantitatively want to match as closely as possible: specifically, we want to minimize the *error* of the output program P from the functional specification \hat{P} , defined as $\text{Er}(P) := \Pr_{x \sim \mathbb{D}}[P(x) \neq \hat{P}(x)]$. (Note that we represent the functional specification as a program.) The postcondition is Boolean, and therefore we always want it to be true. DIGITS is guaranteed to converge whenever the space of solutions satisfying the postcondition is *robust* under small perturbations. The following definition captures this notion of robustness:

Definition 1 (α -Robust Programs). Fix an input distribution \mathbb{D} , a postcondition $post$, and a set of programs \mathcal{P} . For any $P \in \mathcal{P}$ and any $\alpha > 0$, denote the

² Restricting the output to Boolean is required by the algorithm; other output types can be turned into Boolean by rewriting. See, e.g., thermostat example in Sect. 5.

open α -ball centered at P as $B_\alpha(P) = \{P' \in \mathcal{P} \mid \Pr_{x \sim \mathbb{D}}[P(x) \neq P'(x)] < \alpha\}$. We say a program P is α -robust if $\forall P' \in B_\alpha(P). \llbracket P' \rrbracket(\mathbb{D}) \models post$.

We can now state the synthesis problem solved by DIGITS:

Definition 2 (Synthesis Problem). Given an input distribution \mathbb{D} , a set of programs \mathcal{P} , a postcondition $post$, a functional specification $\hat{P} \in \mathcal{P}$, and parameters $\alpha > 0$ and $0 < \varepsilon \leq \alpha$, the synthesis problem is to find a program $P \in \mathcal{P}$ such that $\llbracket P \rrbracket(\mathbb{D}) \models post$ and such that any other α -robust P' has $Er(P) \leq Er(P') + \varepsilon$.

2.2 A Naive DIGITS Algorithm

Algorithm 1 shows a simplified, naive version of DIGITS, which employs a *synthesize-then-verify* approach. The idea of DIGITS is to utilize non-probabilistic synthesis techniques to synthesize a set of programs, and then apply a probabilistic verification step to check if any of the synthesized programs is a solution.

Specifically, this “Naive DIGITS” begins by sampling an appropriate number of inputs from the input distribution and stores them in the set S . Second, it iteratively explores each possible function f that maps the input samples to a Boolean and invokes a synthesis oracle to synthesize a program P that implements f , i.e. that satisfies the set of input-output examples in which each input $x \in S$ is mapped to the output $f(x)$. Naive DIGITS then finds which of the synthesized programs satisfy the postcondition (the set res); we assume that we have access to a probabilistic verifier \mathcal{O}_{ver} to perform these computations. Finally, the algorithm outputs the program in the set res that has the lowest error with respect to the functional specification, once again assuming access to another oracle \mathcal{O}_{err} that can measure the error.

Note that the number of such functions $f : S \rightarrow \{0, 1\}$ is exponential in the size of $|S|$. As a “heuristic” to improve performance, the actual DIGITS algorithm as presented in [1] employs an incremental trie-based search, which we describe (alongside our new algorithm, τ -DIGITS) and analyze in Sect. 3. The naive version described here is, however, sufficient to discuss the convergence properties of the full algorithm.

2.3 Convergence Guarantees

DIGITS is only guaranteed to converge when the program model \mathcal{P} has *finite VC dimension*.³ Intuitively, the VC dimension captures the expressiveness of the set

³ Recall that this is largely a “free” assumption since, again, sketches in our loop-free grammar guaranteedly have finite VC dimension.

```

1 Procedure DIGITS ( $\hat{P}, \mathbb{D}, post, m$ )
2    $S \leftarrow \{x \sim \mathbb{D} \mid i \in [1, \dots, m]\}$ 
3    $progs \leftarrow \emptyset$ 
4   foreach  $f : S \rightarrow \{0, 1\}$  do
5      $P \leftarrow \mathcal{O}_{syn}(\{(x, f(x)) \mid x \in S\})$ 
6     if  $P \neq \perp$  then
7        $progs \leftarrow progs \cup \{P\}$ 
8    $res \leftarrow \{P \in progs \mid$ 
9    $\mathcal{O}_{ver}(P, \mathbb{D}, post)\}$ 
9   return  $\text{argmin}_{P \in res} \{\mathcal{O}_{err}(P)\}$ 
```

Algorithm 1: Naive DIGITS

of ($\{0, 1\}$ -valued) programs \mathcal{P} . Given a set of inputs S , we say that \mathcal{P} *shatters* S iff, for every partition of S into sets $S_0 \sqcup S_1$, there exists a program $P \in \mathcal{P}$ such that (i) for every $x \in S_0$, $P(x) = 0$, and (ii) for every $x \in S_1$, $P(x) = 1$.

Definition 3 (VC Dimension). *The VC dimension of a set of programs \mathcal{P} is the largest integer d such that there exists a set of inputs S with cardinality d that is shattered by \mathcal{P} .*

We define the function $\text{VCCOST}(\varepsilon, \delta, d) = \frac{1}{\varepsilon}(4 \log_2(\frac{2}{\delta}) + 8d \log_2(\frac{13}{\varepsilon}))$ [5], which is used in the following theorem:

Theorem 1 (Convergence). *Assume that there exist an $\alpha > 0$ and program P^* that is α -robust w.r.t. \mathbb{D} and post. Let d be the VC dimension of the set of programs \mathcal{P} . For all bounds $0 < \varepsilon \leq \alpha$ and $\delta > 0$, for every function \mathcal{O}_{syn} , and for any $m \geq \text{VCCOST}(\varepsilon, \delta, k)$, with probability $\geq 1 - \delta$ we have that DIGITS enumerates a program P with $\Pr_{x \sim \mathbb{D}}[P^*(x) \neq P(x)] \leq \varepsilon$ and $\llbracket P \rrbracket(\mathbb{D}) \models \text{post}$.*

To reiterate, suppose P^* is a correct program with small error $\text{Er}(P^*) = k$; the convergence result follows two main points: (i) P^* must be α -robust, meaning every P with $\Pr_{x \sim \mathbb{D}}[P(x) \neq P^*(x)] < \alpha$ must also be correct, and therefore (ii) by synthesizing *any* P such that $\Pr_{x \sim \mathbb{D}}[P(x) \neq P^*(x)] \leq \varepsilon$ where $\varepsilon < \alpha$, then P is a correct program with error $\text{Er}(P)$ within $k \pm \varepsilon$.

2.4 Understanding Convergence

The importance of finite VC dimension is due to the fact that the convergence statement borrows directly from *probably approximately correct (PAC) learning*. We will briefly discuss a core detail of efficient PAC learning that is relevant to understanding the convergence of DIGITS (and, in turn, our analysis of τ -DIGITS in Sect. 4), and refer the interested reader to Kearns and Vazirani’s book [16] for a complete overview. Specifically, we consider the notion of an ε -net, which establishes the approximate-definability of a target program in terms of points in its input space.

Definition 4 (ε -net). *Suppose $P \in \mathcal{P}$ is a target program, and points in its input domain \mathcal{X} are distributed $x \sim \mathbb{D}$. For a fixed $\varepsilon \in [0, 1]$, we say a set of points $S \subset \mathcal{X}$ is an ε -net for P (with respect to \mathcal{P} and \mathbb{D}) if for every $P' \in \mathcal{P}$ with $\Pr_{x \sim \mathbb{D}}[P(x) \neq P'(x)] > \varepsilon$ there exists a witness $x \in S$ such that $P(x) \neq P'(x)$.*

In other words, if S is an ε -net for P , and if P' “agrees” with P on all of S , then P and P' can only differ by at most ε probability mass.

Observe the relevance of ε -nets to the convergence of DIGITS: the synthesis oracle is guaranteed not to “fail” by producing only programs ε -far from some ε -robust P^* if the sample set happens to be an ε -net for P^* . In fact, this observation is exactly the core of the PAC learning argument: having an ε -net exactly guarantees the approximate learnability.

A remarkable result of computational learning theory is that whenever \mathcal{P} has finite VC dimension, the probability that m random samples fail to yield an ε -net

becomes diminishingly small as m increases. Indeed, the given VCCOST function used in Theorem 1 is a dual form of this latter result—that polynomially many samples are sufficient to form an ε -net with high probability.

3 The Efficiency of Trie-Based Search

After providing details on the search strategy employed by DIGITS, we present our theoretical result on the polynomial bound on the number of synthesis queries that DIGITS requires.

3.1 The Trie-Based Search Strategy of DIGITS

Naive DIGITS, as presented in Algorithm 1, performs a very unstructured, exponential search over the output labelings of the sampled inputs—i.e., the possible Boolean functions f in Algorithm 1. In our original paper [1] we present a “heuristic” implementation strategy that incrementally explores the set of possible output labelings using a trie data structure. In this section, we study the complexity of this technique through the lens of computational learning theory and discover the surprising result that DIGITS requires a polynomial number of calls to the synthesizer in the size of the sample set! Our improved search algorithm (Sect. 4) inherits these results.

For the remainder of this paper, we use DIGITS to refer to this incremental version. A full description is necessary for our analysis: Fig. 1 (non-framed rules only) consists of a collection of guarded rules describing the construction of the trie used by DIGITS to incrementally explore the set of possible output labelings. Our improved version, τ -DIGITS (presented in Sect. 4), corresponds to the addition of the framed parts, but without them, the rules describe DIGITS.

Nodes in the trie represent partial output labelings—i.e., functions f assigning Boolean values to only some of the samples in $S = \{x_1, \dots, x_m\}$. Each node is identified by a binary string $\sigma = b_1 \cdots b_k$ (k can be smaller than m) denoting the path to the node from the root. The string σ also describes the partial output-labeling function f corresponding to the node—i.e., if the i -th bit b_i is set to 1, then $f(x_i) = \text{true}$. The set explored represents the nodes in the trie built thus far; for each new node, the algorithm synthesizes a program consistent with the corresponding partial output function (“Explore” rules). The variable depth controls the incremental aspect of the search and represents the maximum length of any σ in explored ; it is incremented whenever all nodes up to that depth have been explored (the “Deepen” rule). The crucial part of the algorithm is that, if no program can be synthesized for the partial output function of a node identified by σ , the algorithm does not need to issue further synthesis queries for the descendants of σ .

Figure 2 shows how DIGITS builds a trie for an example run on the interval programs from Example 1, where we suppose we begin with an incorrect program describing the interval $[0, 0.3]$. Initially, we set the root program to $[0, 0.3]$ (left

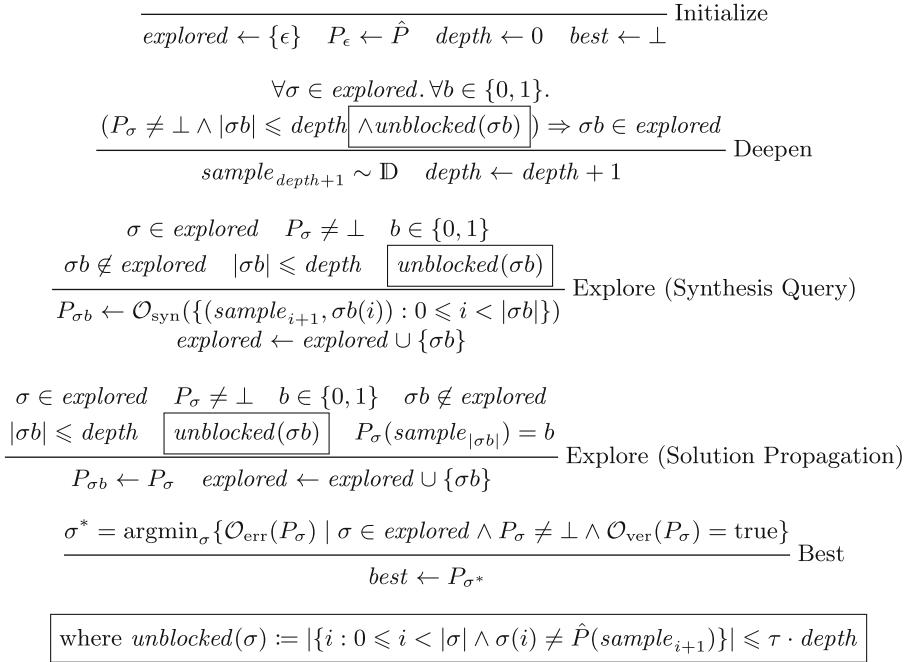


Fig. 1. Full DIGITS description and our new extension, τ -DIGITS, shown in boxes.

figure). The “Deepen” rule applies, so a sample is added to the set of samples—suppose it’s 0.4. “Explore” rules are then applied twice to build the children of the root: the child following the 0 branch needs to map $0.4 \mapsto 0$, which $[0, 0.3]$ already does, thus it is propagated to that child without asking \mathcal{O}_{syn} to perform a synthesis query. For the child following 1, we instead make a synthesis query, using the oracle \mathcal{O}_{syn} , for any value of a such that $[0, a]$ maps $0.4 \mapsto 1$ —suppose it returns the solution $a = 1$, and we associate $[0, 1]$ with this node. At this point we have exhausted depth 1 (middle figure), so “Deepen” once again applies, perhaps adding 0.6 to the sample set. At this depth (right figure), only two calls to \mathcal{O}_{syn} are made: in the case of the call at $\sigma = 01$, there is no value of a that causes both $0.4 \mapsto 0$ and $0.6 \mapsto 1$, so \mathcal{O}_{syn} returns \perp , and we do not try to explore any children of this node in the future. The algorithm continues in this manner until a stopping condition is reached—e.g., enough samples are enumerated.

3.2 Polynomial Bound on the Number of Synthesis Queries

We observed in [1] that the trie-based exploration seems to be efficient in practice, despite potential exponential growth of the number of explored nodes in the trie as the depth of the search increases. The convergence analysis of DIGITS relies on the finite VC dimension of the program model, but VC dimension itself is just a summary of the *growth function*, a function that describes a notion

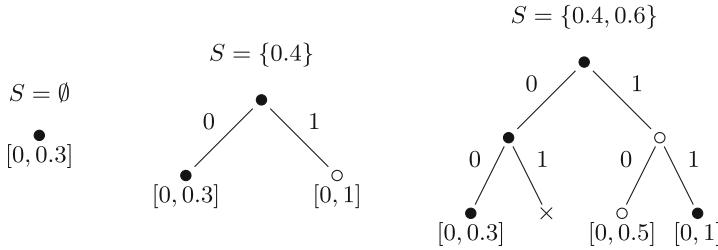


Fig. 2. Example execution of incremental DIGITS on interval programs, starting from $[0, 0.3]$. Hollow circles denote calls to \mathcal{O}_{syn} that yield new programs; the cross denotes a call to \mathcal{O}_{syn} that returns \perp .

of complexity of the set of programs in question. We will see that the growth function much more precisely describes the behavior of the trie-based search; we will then use a classic result from computational learning theory to derive better bounds on the performance of the search. We define the growth function below, adapting the presentation from [16].

Definition 5 (Realizable Dichotomies). *We are given a set \mathcal{P} of programs representing functions from $\mathcal{X} \rightarrow \{0, 1\}$ and a (finite) set of inputs $S \subset \mathcal{X}$. We call any $f : S \rightarrow \{0, 1\}$ a dichotomy of S ; if there exists a program $P \in \mathcal{P}$ that extends f to its full domain \mathcal{X} , we call f a realizable dichotomy in \mathcal{P} . We denote the set of realizable dichotomies as*

$$\Pi_{\mathcal{P}}(S) := \{f : S \rightarrow \{0, 1\} \mid \exists P \in \mathcal{P}. \forall x \in S. P(x) = f(x)\}.$$

Observe that for any (infinite) set \mathcal{P} and any finite set S that $1 \leq |\Pi_{\mathcal{P}}(S)| \leq 2^{|S|}$. We define the growth function in terms of the realizable dichotomies:

Definition 6 (Growth Function). *The growth function is the maximal number of realizable dichotomies as a function of the number of samples, denoted*

$$\hat{\Pi}_{\mathcal{P}}(m) := \max_{\substack{S \subset \mathcal{X}: \\ |S|=m}} \{|\Pi_{\mathcal{P}}(S)|\}.$$

Observe that \mathcal{P} has VC dimension d if and only if d is the largest integer satisfying $\hat{\Pi}_{\mathcal{P}}(d) = 2^d$ (and infinite VC dimension when $\hat{\Pi}_{\mathcal{P}}(m)$ is identically 2^m)—in fact, VC dimension is often defined using this characterization.

Example 2. Consider the set of intervals of the form $[0, a]$ as in Examples 1 and Fig. 2. For the set of two points $S = \{0.4, 0.6\}$, we have that $|\Pi_{[0,a]}(S)| = 3$, since, by example: $a = 0.5$ accepts 0.4 but not 0.6, $a = 0.3$ accepts neither, and $a = 1$ accepts both, thus these three dichotomies are realizable; however, no interval with 0 as a left endpoint can accept 0.6 and not 0.4, thus this dichotomy is not realizable. In fact, for any (finite) set $S \subset [0, 1]$, we have that $|\Pi_{[0,a]}(S)| = |S| + 1$; we then have that $\hat{\Pi}_{[0,a]}(m) = m + 1$.

When DIGITS terminates having used a sample set S , it has considered all the dichotomies of S : the programs it has enumerated exactly correspond to extensions of the realizable dichotomies $\Pi_{\mathcal{P}}(S)$. The trie-based exploration is effectively trying to minimize the number of \mathcal{O}_{syn} queries performed on non-realizable ones, but doing so without explicit knowledge of the full functional behavior of programs in \mathcal{P} . In fact, it manages to stay relatively close to performing queries only on the realizable dichotomies:

Lemma 1. *DIGITS performs at most $|S||\Pi_{\mathcal{P}}(S)|$ synthesis oracle queries. More precisely, let $S = \{x_1, \dots, x_m\}$ be indexed by the depth at which each sample was added: the exact number of synthesis queries is $\sum_{\ell=1}^m |\Pi_{\mathcal{P}}(\{x_1, \dots, x_{\ell-1}\})|$.*

Proof. Let T_d denote the total number of queries performed once depth d is completed. We perform no queries for the root,⁴ thus $T_0 = 0$. Upon completing depth $d - 1$, the realizable dichotomies of $\{x_1, \dots, x_{d-1}\}$ exactly specify the nodes whose children will be explored at depth d . For each such node, one child is skipped due to solution propagation, while an oracle query is performed on the other, thus $T_d = T_{d-1} + |\Pi_{\mathcal{P}}(\{x_1, \dots, x_{d-1}\})|$. Lastly, $|\Pi_{\mathcal{P}}(S)|$ cannot decrease by adding elements to S , so we have that $T_m = \sum_{\ell=1}^m |\Pi_{\mathcal{P}}(\{x_1, \dots, x_{\ell-1}\})| \leq \sum_{\ell=1}^m |\Pi_{\mathcal{P}}(S)| \leq |S||\Pi_{\mathcal{P}}(S)|$. \square

Connecting DIGITS to the realizable dichotomies and, in turn, the growth function allows us to employ a remarkable result from computational learning theory, stating that the growth function for any set exhibits one of two asymptotic behaviors: it is either *identically* 2^m (infinite VC dimension) or dominated by a polynomial! This is commonly called the Sauer-Shelah Lemma [24, 26]:

Lemma 2 (Sauer-Shelah). *If \mathcal{P} has finite VC dimension d , then for all $m \geq d$, $\hat{\Pi}_{\mathcal{P}}(m) \leq \left(\frac{em}{d}\right)^d$; i.e. $\hat{\Pi}_{\mathcal{P}}(m) = O(m^d)$.*

Combining our lemma with this famous one yields a surprising result—that for a fixed set of programs \mathcal{P} with finite VC dimension, the number of oracle queries performed by DIGITS is *guaranteedly polynomial* in the depth of the search, where the degree of the polynomial is determined by the VC dimension:

Theorem 2. *If \mathcal{P} has VC dimension d , then DIGITS performs $O(m^{d+1})$ synthesis-oracle queries.*

In short, the reason an execution of DIGITS *seems* to enumerate a sub-exponential number of programs (as a function of the depth of the search) is because it literally must be polynomial. Furthermore, the algorithm performs oracle queries on *nearly* only those polynomially-many realizable dichotomies.

Example 3. A DIGITS run on the $[0, a]$ programs as in Fig. 2 using a sample set of size m will perform $O(m^2)$ oracle queries, since the VC dimension of these intervals is 1. (In fact, every run of the algorithm on these programs will perform exactly $\frac{1}{2}m(m + 1)$ many queries.)

⁴ We assume the functional specification itself is some $\hat{P} \in \mathcal{P}$ and thus can be used—the alternative is a trivial synthesis query on an empty set of constraints.

4 Property-Directed τ -DIGITS

DIGITS has better convergence guarantees when it operates on larger sets of sampled inputs. In this section, we describe a new optimization of DIGITS that reduces the number of synthesis queries performed by the algorithm so that it more quickly reaches higher depths in the trie, and thus allows to scale to larger samples sets. This optimized DIGITS, called τ -DIGITS, is shown in Fig. 1 as the set of all the rules of DIGITS plus the framed elements. The high-level idea is to skip synthesis queries that are (quantifiably) unlikely to result in optimal solutions. For example, if the functional specification \hat{P} maps every sampled input in S to 0, then the synthesis query on the mapping of every element of S to 1 becomes increasingly likely to result in programs that have maximal distance from \hat{P} as the size of S increases; hence the algorithm could probably avoid performing that query. In the following, we make use of the concept of *Hamming distance* between pairs of programs:

Definition 7 (Hamming Distance). For any finite set of inputs S and any two programs P_1, P_2 , we denote $\text{Hamming}_S(P_1, P_2) := |\{x \in S \mid P_1(x) \neq P_2(x)\}|$ (we will also allow any $\{0, 1\}$ -valued string to be an argument of Hamming_S).

4.1 Algorithm Description

Fix the given functional specification \hat{P} and suppose that there exists an ε -robust solution P^* with (nearly) minimal error $k = \text{Er}(P^*) := \Pr_{x \sim \mathbb{D}}[\hat{P}(x) \neq P^*(x)]$; we would be happy to find *any* program P in P^* 's ε -ball. Suppose we angelically know k a priori, and we thus restrict our search (for each depth m) only to constraint strings (i.e. σ in Fig. 1) that have Hamming distance not much larger than km .

To be specific, we first fix some threshold $\tau \in (k, 1]$. Intuitively, the optimization corresponds to modifying DIGITS to consider only paths σ through the trie such that $\text{Hamming}_S(\hat{P}, \sigma) \leq \tau|S|$. This is performed using the *unblocked* function in Fig. 1. Since we are ignoring certain paths through the trie, we need to ask: *How much does this decrease the probability of the algorithm succeeding?*—It depends on the tightness of the threshold, which we address in Sect. 4.2. In Sect. 4.3, we discuss how to adaptively modify the threshold τ as τ -DIGITS is executing, which is useful when a good τ is unknown a priori.

4.2 Analyzing Failure Probability with Thresholding

Using τ -DIGITS, the choice of τ will affect both (i) how many synthesis queries are performed, and (ii) the likelihood that we *miss* optimal solutions; in this section we explore the latter point.⁵ Interestingly, we will see that all of the analysis is dependent only on parameters directly related to the threshold; notably, none of this analysis is dependent on the complexity of \mathcal{P} (i.e. its VC dimension).

⁵ The former point is a difficult combinatorial question that to our knowledge has no precedent in the computational learning literature, and so we leave it as future work.

If we really want to learn (something close to) a program P^* , then we should use a value of the threshold τ such that $\Pr_{S \sim \mathbb{D}^m}[\text{Hamming}_S(\hat{P}, P^*) \leq \tau m]$ is large—to do so requires knowledge of the distribution of $\text{Hamming}_S(\hat{P}, P^*)$. Recall the *binomial distribution*: for parameters (n, p) , it describes the number of successes in n -many trials of an experiment that has success probability p .

Claim. Fix P and let $k = \Pr_{x \sim \mathbb{D}}[\hat{P}(x) \neq P(x)]$. If S is sampled from \mathbb{D}^m , then $\text{Hamming}_S(\hat{P}, P)$ is binomially distributed with parameters (m, k) .

Next, we will use our knowledge of this distribution to reason about the *failure probability*, i.e. that τ -DIGITS does not preserve the convergence result of DIGITS.

The simplest argument we can make is a union-bound style argument: the thresholded algorithm can “fail” by (i) failing to sample an ε -net, or otherwise (ii) sampling a set on which the optimal solution has a Hamming distance that is not representative of its actual distance. We provide the quantification of this failure probability in the following theorem:

Theorem 3. *Let P^* be a target ε -robust program with $k = \Pr_{x \sim \mathbb{D}}[\hat{P}(x) \neq P^*(x)]$, and let δ be the probability that m samples do not form an ε -net for P^* . If we run the τ -DIGITS with $\tau \in (k, 1]$, then the failure probability is at most $\delta + \Pr[X > \tau m]$ where $X \sim \text{Binomial}(m, k)$.*

In other words, we can use tail probabilities of the binomial distribution to bound the probability that the threshold causes us to “miss” a desirable program we otherwise would have enumerated. Explicitly, we have the following corollary:

Corollary 1. *τ -DIGITS increases failure probability (relative to DIGITS) by at most $\Pr[X > \tau m] = \sum_{i=\lfloor \tau m \rfloor + 1}^m \binom{m}{i} k^i (1-k)^{m-i}$.*

Informally, when m is *not too small*, k is *not too large*, and τ is *reasonably forgiving*, these tail probabilities can be quite small. We can even analyze the asymptotic behavior by using any existing upper bounds on the binomial distribution’s tail probabilities—importantly, the additional error diminishes exponentially as m increases, dependent on the size of τ relative to k .

Corollary 2. *τ -DIGITS increases failure probability by at most $e^{-2m(\tau-k)^2}$.*⁶

Example 4. Suppose $m = 100$, $k = 0.1$, and $\tau = 0.2$. Then the extra failure probability term in Theorem 3 is less than 0.001.

As stated at the beginning of this subsection, the balancing act is to choose τ (i) small enough so that the algorithm is still fast for large m , yet (ii) large enough so that the algorithm is still likely to learn the desired programs. The further challenge is to relax our initial strong assumption that we know the optimal k a priori when determining τ , which we address in the following subsection.

⁶ A more precise (though less convenient) bound is $e^{-m(\tau \ln \frac{\tau}{k} + (1-\tau) \ln \frac{1-\tau}{1-k})}$.

4.3 Adaptive Threshold

Of course, we do not have the angelic knowledge that lets us pick an ideal threshold τ ; the only absolutely sound choice we can make is the trivial $\tau = 1$. Fortunately, we can begin with this choice of τ and *adaptively* refine it as the search progresses. Specifically, every time we encounter a correct program P such that $k = \text{Er}(P)$, we can refine τ to reflect our newfound knowledge that “the best solution has distance of at most k .”

We refer to this refinement as *adaptive τ -DIGITS*. The modification involves the addition of the following rule to Fig. 1:

$$\frac{\text{best} \neq \perp}{\tau \leftarrow g(\mathcal{O}_{\text{err}}(\text{best}))} \text{Refine Threshold (for some } g : [0, 1] \rightarrow [0, 1]\text{)}$$

We can use any (non-decreasing) function g to update the threshold $\tau \leftarrow g(k)$. The simplest choice would be the identity function (which we use in our experiments), although one could use a looser function so as not to over-prune the search. If we choose functions of the form $g(k) = k + b$, then Corollary 2 allows us to make (slightly weak) claims of the following form:

Claim. Suppose the adaptive algorithm completes a search of up to depth m yielding a best solution with error k (so we have the final threshold value $\tau = k + b$). Suppose also that P^* is an optimal ε -robust program at distance $k - \eta$. The optimization-added failure probability (as in Corollary 1) for a run of (non-adaptive) τ -DIGITS completing depth m and using this τ is at most $e^{-2m(b+\eta)^2}$.

5 Evaluation

Implementation. In this section, we evaluate our new algorithm τ -DIGITS (Fig. 1) and its adaptive variant (Sect. 4.3) against DIGITS (i.e., τ -DIGITS with $\tau = 1$). Both algorithms are implemented in Python and use the SMT solver Z3 [8] to implement a sketch-based synthesizer \mathcal{O}_{syn} . We employ statistical verification for \mathcal{O}_{ver} and \mathcal{O}_{err} : we use Hoeffding’s inequality for estimating probabilities in *post* and *Er*. Probabilities are computed with 95% confidence, leaving our oracles potentially unsound.

Research Questions. Our evaluation aims to answer the following questions:

- RQ1** Is adaptive τ -DIGITS more effective/precise than τ -DIGITS?
- RQ2** Is τ -DIGITS more effective/precise than DIGITS?
- RQ3** Can τ -DIGITS solve challenging synthesis problems?

We experiment on three sets of benchmarks: (*i*) synthetic examples for which the optimal solutions can be computed analytically (Sect. 5.1), (*ii*) the set of benchmarks considered in the original DIGITS paper (Sect. 5.2), (*iii*) a variant of the thermostat-controller synthesis problem presented in [7] (Sect. 5.3).

5.1 Synthetic Benchmarks

We consider a class of synthetic programs for which we can compute the optimal solution exactly; this lets us compare the results of our implementation to an ideal baseline. Here, the program model \mathcal{P} is defined as the set of axis-aligned hyperrectangles within $[-1, 1]^d$ ($d \in \{1, 2, 3\}$) and the VC dimension is $2d$, and the input distribution \mathbb{D} is such that inputs are distributed uniformly over $[-1, 1]^d$. We fix some probability mass $b \in \{0.05, 0.1, 0.2\}$ and define the benchmarks so that the best error for a correct solution is exactly b (for details, see [9]).

We run our implementation using thresholds $\tau \in \{0.07, 0.15, 0.3, 0.5, 1\}$, omitting those values for which $\tau < b$; additionally, we also consider an adaptive run where τ is initialized as the value 1, and whenever a new best solution is enumerated with error k , we update $\tau \leftarrow k$. Each combination of parameters was run for a period of 2 min. Figure 3 fixates on $d = 1$, $b = 0.1$ and shows each of the following as a function of time: (i) the depth completed by the search (i.e. the current size of the sample set), and (ii) the best solution found by the search. (See our full version of the paper [9] for other configurations of (d, b) .)

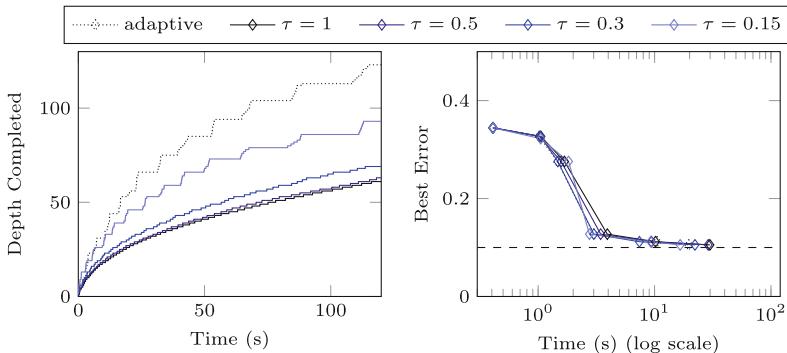


Fig. 3. Synthetic hyperrectangle problem instance with parameters $d = 1$, $b = 0.1$.

By studying Fig. 3 we see that the adaptive threshold search performs at least as well as the tight thresholds fixed a priori because reasonable solutions are found early. In fact, all search configurations find solutions very close to the optimal error (indicated by the horizontal dashed line). Regardless, they reach different depths, and *the main advantage of reaching large depths concerns the strength of the optimality guarantee*. Note, also, that small τ values are necessary to see improvements in the completed depth of the search. Indeed, the discrepancy between the depth-versus-time functions diminishes drastically for the problem instances with larger values of b (See our full version of the paper [9]); the gains of the optimization are contingent on the existence of correct solutions *close* to the functional specification.

Findings (RQ1): τ -DIGITS does tend to find *reasonable* solutions at early depths and near-optimal solutions at later depths, thus adaptive τ -DIGITS is more effective than τ -DIGITS, and we use it throughout our remaining experiments.

5.2 Original DIGITS Benchmarks

The original DIGITS paper [1] evaluates on a set of 18 repair problems of varying complexity. The functional specifications are machine-learned decision trees and support vector machines, and each search space \mathcal{P} involves the set of programs formed by replacing some number of real-valued constants in the program with holes. The postcondition is a form of *algorithmic fairness*—e.g., the program should output true on inputs of type *A* as often as it does on inputs of type *B* [11]. For each such repair problem, we run both DIGITS and adaptive τ -DIGITS (again, with initial $\tau = 1$ and the identity refinement function). Each benchmark is run for 10 min, where the same sample set is used for both algorithms.

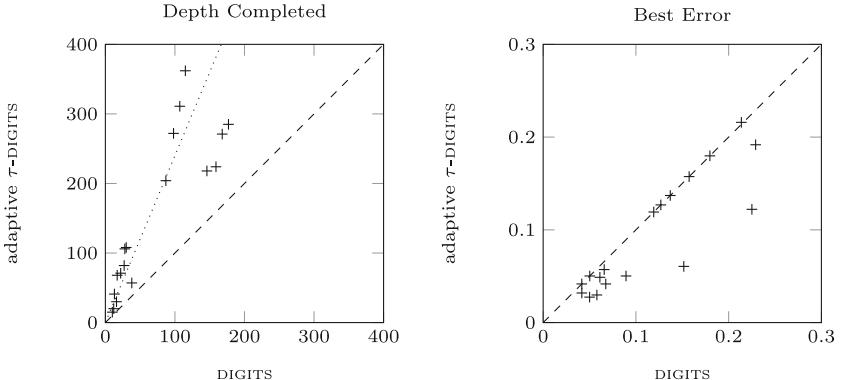


Fig. 4. Improvement of using adaptive τ -DIGITS on the original DIGITS benchmarks. Left: the dotted line marks the 2.4 \times average increase in depth.

Figure 4 shows, for each benchmark, (*i*) the largest sample set size completed by adaptive τ -DIGITS versus DIGITS (left—above the diagonal line indicates adaptive τ -DIGITS reaches further depths), and (*ii*) the error of the best solution found by adaptive τ -DIGITS versus DIGITS (right—below the diagonal line indicates adaptive τ -DIGITS finds better solutions). We see that adaptive τ -DIGITS reaches further depths on every problem instance, many of which are substantial improvements, and that it finds better solutions on 10 of the 18 problems. For those which did not improve, either the search was already deep enough that DIGITS was able to find near-optimal solutions, or the complexity of the synthesis queries is such that the search is still constrained to small depths.

Findings (RQ2): Adaptive τ -DIGITS can find better solutions than those found by DIGITS and can reach greater search depths.

5.3 Thermostat Controller

We challenge adaptive τ -DIGITS with the task of synthesizing a thermostat controller, borrowing the benchmark from [7]. The input to the controller is the initial temperature of the environment; since the world is uncertain, there is a specified probability distribution over the temperatures. The controller itself is a program sketch consisting primarily of a single main loop: iterations of the loop correspond to timesteps, during which the synthesized parameters dictate an incremental update made by the thermostat based on the current temperature. The loop runs for 40 iterations, then terminates, returning the absolute value of the difference between its final actual temperature and the target temperature.

The postcondition is a Boolean probabilistic correctness property intuitively corresponding to controller safety, e.g. with high probability, the temperature should never exceed certain thresholds. In [7], there is a quantitative objective in the form of minimizing the expected value $E[|actual - target|]$ —our setting does not admit optimizing with respect to expectations, so we must modify the problem. Instead, we fix some value N ($N \in \{2, 4, 8\}$) and have the program return 0 when $|actual - target| < N$ and 1 otherwise. Our quantitative objective is to minimize the error from the constant-zero functional specification $\hat{P}(x) := 0$ (i.e. the actual temperature always gets close enough to the target). The full specification of the controller is provided in the full version of our paper [9].

We consider variants of the program where the thermostat runs for fewer timesteps and try loop unrollings of size $\{5, 10, 20, 40\}$. We run each benchmark for 10 min: the final completed search depths and best error of solutions are shown in Fig. 5. For this particular experiment, we use the SMT solver CVC4 [3] because it performs better than Z3 on the occurring SMT instances.

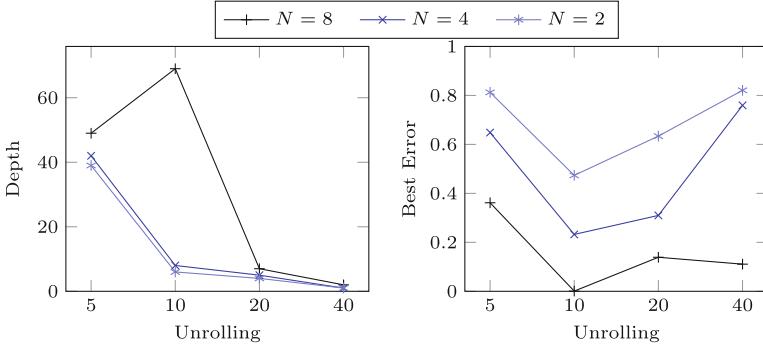


Fig. 5. Thermostat controller results.

As we would expect, for larger values of N it is “easier” for the thermostat to reach the target temperature threshold and thus the quality of the best solution increases in N . However, with small unrollings (i.e. 5) the synthesized controllers do not have enough iterations (time) to modify the temperature enough for the

probability mass of extremal temperatures to reach the target: as we increase the number of unrollings to 10, we see that better solutions can be found since the set of programs are capable of stronger behavior.

On the other hand, the completed depth of the search plummets as the unrolling increases due to the complexity of the \mathcal{O}_{syn} queries. Consequently, for 20 and 40 unrollings, adaptive τ -DIGITS synthesizes worse solutions because it cannot reach the necessary depths to obtain better guarantees.

One final point of note is that for $N = 8$ and 10 unrollings, it seems that there is a sharp spike in the completed depth. However, this is somewhat artificial: because $N = 8$ creates a very lenient quantitative objective, an early \mathcal{O}_{syn} query happens to yield a program with an error less than 10^{-3} . Adaptive τ -DIGITS then updates $\tau \leftarrow \approx 10^{-3}$ and skips most synthesis queries.

Findings (RQ3): Adaptive τ -DIGITS can synthesize small variants of a complex thermostat controller, but cannot solve variants with many loop iterations.

6 Related Work

Synthesis and Probability. Program synthesis is a mature area with many powerful techniques. The primary focus is on synthesis under Boolean constraints, and probabilistic specifications have received less attention [1, 7, 17, 19]. We discuss the works that are most related to ours.

DIGITS [1] is the most relevant work. First, we show for the first time that DIGITS only requires a number of synthesis queries polynomial in the number of samples. Second, our adaptive τ -DIGITS further reduces the number of synthesis queries required to solve a synthesis problem without sacrificing correctness.

The technique of *smoothed proof search* [7] approximates a combination of functional correctness and maximization of an expected value as a smooth, continuous function. It then uses numerical methods to find a local optimum of this function, which translates to a synthesized program that is likely to be correct and locally maximal. The benchmarks described in Sect. 5.3 are variants of benchmarks from [7]. Smoothed proof search can minimize expectation; τ -DIGITS minimizes probability only. However, unlike τ -DIGITS, smoothed proof search lacks formal convergence guarantees and cannot support the rich probabilistic postconditions we support, e.g., as in the fairness benchmarks.

Works on synthesis of probabilistic programs are aimed at a different problem [6, 19, 23]: that of synthesizing a generative model of data. For example, Nori et al. [19] use sketches of probabilistic programs and complete them with a stochastic search. Recently, Saad et al. [23] synthesize an ensemble of probabilistic programs for learning Gaussian processes and other models.

Kúcera et al. [17] present a technique for automatically synthesizing program transformations that introduce uncertainty into a given program with the goal of satisfying given privacy policies—e.g., preventing information leaks. They leverage the specific structure of their problem to reduce it to an SMT constraint solving problem. The problem tackled in [17] is orthogonal to the one targeted in this paper and the techniques are therefore very different.

Stochastic Satisfiability. Our problem is closely related to E-MAJSAT [18], a special case of *stochastic satisfiability* (ssAT) [20] and a means for formalizing probabilistic planning problems. E-MAJSAT is of NP^{PP} complexity. An E-MAJSAT formula has deterministic and probabilistic variables. The goal is to find an assignment of deterministic variables such that the probability that the formula is satisfied is above a given threshold. Our setting is similar, but we operate over complex program statements and have an additional optimization objective (i.e., the program should be close to the functional specification). The deterministic variables in our setting are the holes defining the search space; the probabilistic variables are program inputs.

Acknowledgements. We thank Shuchi Chawla, Yingyu Liang, Jerry Zhu, the entire fairness reading group at UW-Madison, and Nika Haghtalab for all of the detailed discussions. This material is based upon work supported by the National Science Foundation under grant numbers 1566015, 1704117, and 1750965.

References

1. Albarghouthi, A., D’Antoni, L., Drews, S.: Repairing decision-making programs under uncertainty. In: Majumdar, R., Kunčak, V. (eds.) Computer Aided Verification, pp. 181–200. Springer International Publishing, Cham (2017)
2. Barowy, D.W., Gulwani, S., Hart, T., Zorn, B.G.: Flashrelate: extracting relational data from semi-structured spreadsheets using examples. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, 15–17 June 2015, pp. 218–228 (2015). <https://doi.org/10.1145/2737924.2737952>
3. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_14
4. Bastani, O., Sharma, R., Aiken, A., Liang, P.: Synthesizing program input grammars. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, 18–23 June 2017, pp. 95–110 (2017). <https://doi.org/10.1145/3062341.3062349>
5. Blumer, A., Ehrenfeucht, A., Haussler, D., Warmuth, M.K.: Learnability and the vapnik-chervonenkis dimension. J. ACM (JACM) **36**(4), 929–965 (1989)
6. Chasins, S., Phothilimthana, P.M.: Data-driven synthesis of full probabilistic programs. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 279–304. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_14
7. Chaudhuri, S., Clochard, M., Solar-Lezama, A.: Bridging boolean and quantitative synthesis using smoothed proof search. In: POPL, vol. 49, pp. 207–220. ACM (2014)
8. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
9. Drews, S., Albarghouthi, A., D’Antoni, L.: Efficient synthesis with probabilistic constraints (2019). <http://arxiv.org/abs/1905.08364>
10. El-Hassany, A., Tsankov, P., Vanbever, L., Vechev, M.: Network-wide configuration synthesis. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10427, pp. 261–281. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63390-9_14

11. Feldman, M., Friedler, S.A., Moeller, J., Scheidegger, C., Venkatasubramanian, S.: Certifying and removing disparate impact. In: Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 259–268. ACM (2015)
12. Goldberg, P.W., Jerrum, M.: Bounding the vapnik-chervonenkis dimension of concept classes parameterized by real numbers. *Mach. Learn.* **18**(2–3), 131–148 (1995). <https://doi.org/10.1007/BF00993408>
13. Gulwani, S.: Automating string processing in spreadsheets using input-output examples. In: Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, 26–28 January 2011, pp. 317–330 (2011). <https://doi.org/10.1145/1926385.1926423>
14. Gulwani, S.: Program synthesis. In: Software Systems Safety, pp. 43–75 (2014). <https://doi.org/10.3233/978-1-61499-385-8-43>
15. Gulwani, S.: Programming by examples - and its applications in data wrangling. In: Dependable Software Systems Engineering, pp. 137–158 (2016). <https://doi.org/10.3233/978-1-61499-627-9-137>
16. Kearns, M.J., Vazirani, U.V.: An Introduction to Computational Learning Theory. MIT Press, Cambridge (1994)
17. Kučera, M., Tsankov, P., Gehr, T., Guarnieri, M., Vechev, M.: Synthesis of probabilistic privacy enforcement. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, pp. 391–408. ACM, New York (2017). <https://doi.org/10.1145/3133956.3134079>
18. Littman, M.L., Goldsmith, J., Mundhenk, M.: The computational complexity of probabilistic planning. *J. Artif. Intell. Res.* **9**, 1–36 (1998)
19. Nori, A.V., Ozair, S., Rajamani, S.K., Vijaykeerthy, D.: Efficient synthesis of probabilistic programs. *SIGPLAN Not.* **50**(6), 208–217 (2015). <https://doi.org/10.1145/2813885.2737982>
20. Papadimitriou, C.H.: Games against nature. *J. Comput. Syst. Sci.* **31**(2), 288–301 (1985)
21. Polozov, O., Gulwani, S.: Flashmeta: a framework for inductive program synthesis. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, 25–30 October 2015, pp. 107–126 (2015). <https://doi.org/10.1145/2814270.2814310>
22. Raza, M., Gulwani, S.: Automated data extraction using predictive program synthesis. In: Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, 4–9 February 2017, San Francisco, California, USA, pp. 882–890 (2017). <http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/15034>
23. Saad, F.A., Cusumano-Towner, M.F., Schaechtle, U., Rinard, M.C., Mansinghka, V.K.: Bayesian synthesis of probabilistic programs for automatic data modeling. *Proc. ACM Program. Lang.* **3**(POPL), 37 (2019)
24. Sauer, N.: On the density of families of sets. *J. Comb. Theory, Seri. A* **13**(1), 145–147 (1972)
25. Schkufza, E., Sharma, R., Aiken, A.: Stochastic program optimization. *Commun. ACM* **59**(2), 114–122 (2016). <https://doi.org/10.1145/2863701>
26. Shelah, S.: A combinatorial problem; stability and order for models and theories in infinitary languages. *Pac. J. Math.* **41**(1), 247–261 (1972)
27. Solar-Lezama, A.: Program Synthesis by Sketching. Ph.D. thesis, Berkeley, CA, USA (2008), aAI3353225

28. Srinivasan, V., Reps, T.W.: Synthesis of machine code from semantics. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, 15–17 June 2015, pp. 596–607 (2015). <https://doi.org/10.1145/2737924.2737960>
29. Subramanian, K., D’Antoni, L., Akella, A.: Genesis: synthesizing forwarding tables in multi-tenant networks. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, 18–20 January 2017, pp. 572–585 (2017). <http://dl.acm.org/citation.cfm?id=3009845>
30. Wang, X., Gulwani, S., Singh, R.: FIDEX: filtering spreadsheet data using examples. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, 30 October - 4 November 2016, pp. 195–213 (2016). <https://doi.org/10.1145/2983990.2984030>

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter’s Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter’s Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Membership-Based Synthesis of Linear Hybrid Automata

Miriam García Soto^(✉), Thomas A. Henzinger, Christian Schilling, and Luka Zelezniak

IST Austria, Klosterneuburg, Austria
{miriam.garciasoto,tah,christian.schilling,
luka.zelezniak}@ist.ac.at



Abstract. We present two algorithmic approaches for synthesizing linear hybrid automata from experimental data. Unlike previous approaches, our algorithms work without a template and generate an automaton with nondeterministic guards and invariants, and with an arbitrary number and topology of modes. They thus construct a succinct model from the data and provide formal guarantees. In particular, (1) the generated automaton can reproduce the data up to a specified tolerance and (2) the automaton is tight, given the first guarantee. Our first approach encodes the synthesis problem as a logical formula in the theory of linear arithmetic, which can then be solved by an SMT solver. This approach minimizes the number of modes in the resulting model but is only feasible for limited data sets. To address scalability, we propose a second approach that does not enforce to find a minimal model. The algorithm constructs an initial automaton and then iteratively extends the automaton based on processing new data. Therefore the algorithm is well-suited for online and synthesis-in-the-loop applications. The core of the algorithm is a membership query that checks whether, within the specified tolerance, a given data set can result from the execution of a given automaton. We solve this membership problem for linear hybrid automata by repeated reachability computations. We demonstrate the effectiveness of the algorithm on synthetic data sets and on cardiac-cell measurements.

Keywords: Synthesis · Linear hybrid automaton · Membership

1 Introduction

Natural sciences pursue to understand the mechanisms of real systems and to make this understanding accessible. Achieving these two goals requires observation, analysis, and modeling of the system. Typically, physical components of a

This research was supported in part by the Austrian Science Fund (FWF) under grants S11402-N23 (RiSE/SHiNE) and Z211-N23 (Wittgenstein Award) and the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 754411.

© The Author(s) 2019

I. Dillig and S. Tasiran (Eds.): CAV 2019, LNCS 11561, pp. 297–314, 2019.

https://doi.org/10.1007/978-3-030-25540-4_16

system evolve continuously in real time, while the system may switch among a finite set of discrete states. This applies to cyber-physical systems but also to purely analog systems; e.g., an animal's hunger affects its movement. A proper formalism for modeling such types of systems with mixed discrete-continuous behavior is a hybrid automaton [11]. Unlike black-box models such as neural networks, hybrid automata are easy to interpret by humans. However, designing such models is a time-intensive and error-prone process, usually conducted by an expert who analyzes the experimental data and makes decisions.

In this paper, we propose two automatic approaches for synthesizing a linear hybrid automaton [1] from experimental data. The approaches provide two main properties. The first property is *soundness*, which ensures that the generated model has enough executions: these executions approximate the given data up to a predefined accuracy. The second property is *precision*, which ensures that the generated model does not have too many executions. The behavior of a hybrid automaton is constrained by so-called invariants and guards. *Precision* expresses that the boundaries of these invariants and guards are witnessed by the data, which indicates that the constraints cannot be made tighter. Moreover, the proposed synthesis algorithm is *complete* for a general class of linear hybrid automata, i.e., the algorithm can synthesize any given model from this class.

The first approach reduces the synthesis problem to a satisfiability question for a linear-arithmetic formula. The formula allows us to encode a minimality constraint (namely in the number of so-called modes) on the resulting model. This approach is, however, not scalable, which motivates our second approach. Our second approach follows an iterative model-adaptation scheme. Apart from scalability advantages, this *online* algorithm is thus also well-suited for synthesis-in-the-loop applications.

After constructing an initial model, the second approach iteratively improves and expands the model by considering new experiments. After each iteration, the model will capture all behaviors exhibited in the previous experiments. Given an automaton and new experimental data, the algorithm proceeds as follows. First we ask whether the current automaton already captures the data. We pose this question as a membership query for a piecewise-linear function in the set of executions of the automaton. For the membership query, we present an algorithm based on reachability inside a tube around the function. If the data is not captured, we need to modify the automaton accordingly by adding behavior. We first try to relax the above-mentioned invariants and guards, which we reduce to another membership query. If that query is negative as well, we choose a path in the automaton that closely resembles the given data and then modify the automaton along that path by also adding new discrete structure (called modes and transitions). This modification step is again guided by membership queries to identify the aspects of the model that require improvement and expansion.

As the main contributions, (1) we present an online algorithm for automatic synthesis of linear hybrid automata from data that is *sound*, i.e., guarantees that the generated model approximates the data up to a user-defined threshold, *precise*, i.e., the generated model is tight, and *complete* for a general class of

models (2) we solve the membership problem of a piecewise-linear function in a linear hybrid automaton. This is a critical step in our synthesis algorithm.

Related Work. The synthesis of hybrid systems was initially studied in control theory under the term *identification*, mainly focused on (discrete-time) switched autoregressive exogenous (SARX) and piecewise-affine autoregressive exogenous (PWARX) models [7, 18]. SARX models constitute a subclass of linear hybrid automata with deterministic switching behavior. PWARX models are specific SARX models where the mode invariants form a state-space partition. Fixing the number of modes, the identification problem from input-output data can be solved algebraically by inferring template parameters. However, in contrast to linear hybrid automata, the lack of nondeterminism and the underlying assumption that there is no hidden state (mode) limits the applicability of these models. An algorithm by Bemporad et al. constructs a PWARX model that satisfies a *global* error bound [5]. Ozay presents an algorithm for SARX models where the switching is purely time-triggered [17]. There also exist a few *online* algorithms for the recursive synthesis of PWARX models based on pattern recognition [19] or lifting to a high-dimensional identification problem for ARX models [10, 22].

Synthesis is also known as *process mining*, and as *learning models from traces*; the latter refers to approaches based on learning finite-state machines [3] or other machine-learning techniques. More recently, synthesis of hybrid automaton models has gained attention. All existing approaches that we are aware of have structural restrictions of some sort, which we describe below. We synthesize, for the first time, a general class of linear hybrid automata which (1) allows nondeterminism to capture many behaviors by a *concise* representation and (2) provides formal soundness and precision guarantees. The algorithm is also the first *online* synthesis approach for linear hybrid automata.

The general synthesis problem for hybrid automata is hard: for deterministic timed automata (a subclass of linear hybrid automata with globally identical continuous dynamics), one may already require data of exponential length [21]. The approach by Niggemann et al. constructs an automaton with acyclic discrete structure [16], while the approach by Grosu et al., intended to model purely periodic behavior, constructs a cyclic-linear hybrid automaton whose discrete structure consists of a loop [8]. Ly and Lipson use symbolic regression to infer a non-linear hybrid automaton [14]. However, their model neither contains state variables (i.e., the model is purely input-driven, comparable to the SARX model) nor invariants, and the number of modes needs to be fixed in advance. Medhat et al. describe an abstract framework, based on heuristics, to learn linear hybrid automata from input/output traces [15]. They first employ Angluin’s algorithm for learning a finite-state machine [3], which serves as the discrete structure of the hybrid automaton, before they decorate the automaton with continuous dynamics. This strict separation inherently makes their approach offline. The work by Summerville et al. based on least-squares regression requires an exhaustive construction of all possible models for later optimizing a cost function over all of them [20]. Lamrani et al. learn a completely deterministic model with urgent transitions using ideas from information theory [12].

2 Preliminaries

Sets. Let \mathbb{R} , $\mathbb{R}_{\geq 0}$, and \mathbb{N} denote the set of real numbers, non-negative real numbers, and natural numbers, respectively. We write \mathbf{x} for points (x_1, \dots, x_n) in \mathbb{R}^n . Let $\text{cpoly}(n)$ be the set of compact and convex polyhedral sets over \mathbb{R}^n . A set $X \in \text{cpoly}(n)$ is characterized by its set of vertices $\text{vert}(X)$. For a set of points Y , $\text{chull}(Y) \in \text{cpoly}(n)$ denotes the convex hull. Given a set $X \in \text{cpoly}(n)$ and $\varepsilon \in \mathbb{R}_{\geq 0}$, we define the ε -bloating of X as $[X]_\varepsilon := \{\mathbf{x} \in \mathbb{R}^n \mid \exists \mathbf{x}_0 \in X : \|\mathbf{x} - \mathbf{x}_0\| \leq \varepsilon\} \in \text{cpoly}(n)$, where $\|\cdot\|$ is the infinity norm. Given an interval $I = [l, u] \in \text{cpoly}(1)$, $\text{lb}(I) = l$ and $\text{ub}(I) = u$ denote its lower and upper bound.

Functions and Sequences. Given a function f , let $\text{dom}(f)$ resp. $\text{img}(f)$ denote its domain resp. image. Let $f|_A$ denote the restriction of f to domain $A \subseteq \text{dom}(f)$. We define a *distance* between functions f and g with the same domain and codomain by $d(f, g) := \max_{t \in \text{dom}(f)} \|f(t) - g(t)\|$. A *sequence* of length m is a function $s : D \rightarrow A$ over an ordered finite domain $D = \{i_1, \dots, i_m\} \subseteq \mathbb{N}$ and a set A , and we write $\text{len}(s)$ to denote the length of s . A sequence s is also represented by enumerating its elements, as in $s(i_1), \dots, s(i_m)$.

Affine and Piecewise-Linear Functions. An *affine piece* is a function $p : I \rightarrow \mathbb{R}^n$ over an interval $I = [t_0, t_1] \subseteq \mathbb{R}$ defined as $p(t) = \mathbf{a}t + \mathbf{b}$ where $\mathbf{a}, \mathbf{b} \in \mathbb{R}^n$. Given an affine piece p , $\text{init}(p)$ denotes the start point $p(t_0)$, $\text{end}(p)$ denotes the end point $p(t_1)$, and $\text{slope}(p)$ denotes the slope \mathbf{a} . We call two affine pieces p and p' *adjacent* if $\text{end}(p) = \text{init}(p')$ and $\text{ub}(\text{dom}(p)) = \text{lb}(\text{dom}(p'))$. For $m \in \mathbb{N}$, an m -piecewise-linear (m -PWL) function $f : I \rightarrow \mathbb{R}^n$ over interval $I = [0, T] \subseteq \mathbb{R}$ consists of m affine pieces p_1, \dots, p_m , such that $I = \cup_{1 \leq j \leq m} \text{dom}(p_j)$, $f(t) = p_j(t)$ for $t \in \text{dom}(p_j)$, and for every $1 < j \leq m$ we have $\text{end}(p_{j-1}) = \text{init}(p_j)$. We show a 3-PWL function in Fig. 1 on the left. Let $\text{pieces}(f)$ denote the set of affine pieces of f . We refer to f and the sequence p_1, \dots, p_m interchangeably and write “PWL function” if m is clear from the context. A *kink* of a PWL function is the point between two adjacent pieces. Given a PWL function $f : I \rightarrow \mathbb{R}^n$ and a value $\varepsilon \in \mathbb{R}_{\geq 0}$, the ε -tube of f is the function $\text{tube}_{f, \varepsilon} : I \rightarrow \text{cpoly}(n)$ such that $\text{tube}_{f, \varepsilon}(t) = [f(t)]_\varepsilon$.

Graphs. A *graph* is a pair (V, E) of a finite set V and a relation $E \subseteq V \times V$. A *path* π in (V, E) is a sequence v_1, \dots, v_m with $(v_{j-1}, v_j) \in E$ for $1 < j \leq m$.

Hybrid Automata. We consider a particular class of hybrid automata [1, 11].

Definition 1. A n -dimensional linear hybrid automaton (LHA) is a tuple $\mathcal{H} = (Q, E, X, \text{Flow}, \text{Inv}, \text{Grd})$, where (1) Q is a finite set of modes, (2) $E \subseteq Q \times Q$ is a transition relation, (3) $X = \mathbb{R}^n$ is the continuous state-space, (4) $\text{Flow} : Q \rightarrow \mathbb{R}^n$ is the flow function, (5) $\text{Inv} : Q \rightarrow \text{cpoly}(n)$ is the invariant function, and (6) $\text{Grd} : E \rightarrow \text{cpoly}(n)$ is the guard function

We sometimes annotate the elements of LHA \mathcal{H} by a subscript, as in $Q_{\mathcal{H}}$ for the set of modes. We refer to $(Q_{\mathcal{H}}, E_{\mathcal{H}})$ as the *graph of LHA* \mathcal{H} .

An LHA evolves continuously according to the flow function in each mode. The behavior starts in some mode $q \in Q$ and some continuous state $\mathbf{x} \in \text{Inv}(q)$.

For every mode $q \in Q$, the continuous evolution follows the differential equation $\dot{\mathbf{x}} = \text{Flow}(q)$ while satisfying the invariant $\text{Inv}(q)$. The behavior can switch from one mode q_1 to another mode q_2 if there is a transition $(q_1, q_2) \in E$ and the guard $\text{Grd}((q_1, q_2))$ is satisfied. During a switch, the continuous state does not change. This type of system is sometimes called a switched linear hybrid system [13].

Definition 2. Given an n -dimensional LHA $\mathcal{H} = (Q, E, X, \text{Flow}, \text{Inv}, \text{Grd})$, an execution σ is a triple $\sigma = (\mathcal{I}, \gamma, \delta)$, where \mathcal{I} is a sequence of consecutive intervals $[t_0, t_1], [t_1, t_2], \dots, [t_{m-1}, t_m]$ with $[\mathcal{I}] = \cup_{0 \leq j < m} [t_j, t_{j+1}]$, and $\gamma : [\mathcal{I}] \rightarrow \mathbb{R}^n$ and $\delta : \{1, \dots, m\} \rightarrow Q$ are functions with the following restrictions:

- for all $1 \leq j < m$, $\gamma(t) \in \text{Inv}(\delta(j))$ for $t \in \mathcal{I}(j)$ and $\dot{\gamma}(t') = \text{Flow}(\delta(j))$ for all t' in the interior of $\mathcal{I}(j)$, i.e., $\gamma|_{\mathcal{I}(j)}$ is an affine function satisfying the invariant and following the flow, and
- for all $1 \leq j < m$, $(\delta(j), \delta(j+1)) \in E$ and $\gamma(t) \in \text{Grd}((\delta(j), \delta(j+1)))$ where $t = \text{ub}(\mathcal{I}(j))$, i.e., if a transition is taken, then the guard is satisfied.

We denote the set of all executions of \mathcal{H} by $\text{exec}(\mathcal{H})$. Given an LHA \mathcal{H} , we say that an execution σ follows a path π in \mathcal{H} , that is, in the graph $(Q_{\mathcal{H}}, E_{\mathcal{H}})$, denoted as $\sigma \xrightarrow{\mathcal{H}} \pi$, if $\text{len}(\mathcal{I}) = \text{len}(\pi)$ and $\delta(j) = \pi(j)$ for every $0 \leq j < \text{len}(\mathcal{I})$.

From Time-series Data to PWL Functions. Experimental data typically comes as *time series*, i.e., data is only available at sampled points in time. A time series is a sampling $s : D \rightarrow \mathbb{R}^n$ over a finite time domain $D \subseteq [0, T]$. Since the LHA model features piecewise-linear executions, we focus on piecewise-linear approximation of the data. PWL functions can approximate any continuous behavior with arbitrary precision. There are different yet valid choices for approximating data. For a single time series, linear interpolation gives a perfect fit, but contains many kinks; other algorithms minimize the number of kinks for a given error bound [6, 9]. One can preprocess multiple time series into a single PWL function using, e.g., linear regression. In this paper, we leave the choice of abstraction open and assume that the input is given as PWL functions.

3 Synthesis of Linear Hybrid Automata

In this section, we specify the synthesis problem, consider two different specifications, synchronous and asynchronous, and present the automated approach for solving the synchronous problem. The overall goal is to synthesize a linear hybrid automaton from a set of PWL functions such that the automaton *captures* the behavior described by each of the PWL functions up to a bound ε .

Definition 3 (Soundness). Given a PWL function f and a value $\varepsilon \in \mathbb{R}_{\geq 0}$, we say that an LHA \mathcal{H} ε -captures f if there exists an execution $\sigma = (\mathcal{I}, \gamma, \delta)$ in $\text{exec}(\mathcal{H})$ with $d(f, \gamma) \leq \varepsilon$.

The value ε quantifies the acceptable deviation of an execution's continuous function γ from the PWL function f . For $\varepsilon = 0$, γ must precisely follow f . A straightforward formulation of the problem we want to solve is the following.

Problem 1 (Synthesis). Given a finite set of PWL functions \mathcal{F} and $\varepsilon \in \mathbb{R}_{\geq 0}$, construct an LHA \mathcal{H} that ε -captures every function $f \in \mathcal{F}$.

Observe that this problem is not well-posed, as it can be satisfied by an automaton that exhibits an excessive amount of behavior. Hence our second goal for the synthesis algorithm is to ensure constraints on the automaton's size. We start with the synthesis of an LHA with minimal number of modes.

3.1 Synchronous Switching Specification

For now, we require that the executions in the LHA switch *synchronously* with the given PWL functions. Under this assumption, we tackle a refinement of Problem 1:

Problem 2 (Synchronous synthesis). Given a finite set of PWL functions \mathcal{F} and a value $\varepsilon \in \mathbb{R}_{\geq 0}$, construct an LHA \mathcal{H} that ε -captures every function $f \in \mathcal{F}$ synchronously, and furthermore require that \mathcal{H} has the minimal number of modes.

In the following, we present an algorithm to solve Problem 2. The idea is, given a PWL function f , to synthesize an execution σ that is ε -close to f . Recall that the continuous function γ of an execution is essentially just another PWL function. Any LHA that contains the execution σ has to comprise a mode for each different slope in γ . Thus a minimal number of modes can be achieved by minimizing the number of different slopes in γ . By fixing a number of different slopes, we encode the existence of γ as a logical formula $\phi_{f,\varepsilon}$, which will be satisfiable if and only if there exists a suitable function γ .

Let m be the number of affine pieces p_1, \dots, p_m in f with $\text{dom}(p_j) = [t_{j-1}, t_j]$ for $1 \leq j \leq m$. We refer to the time instants t_j as the switching times of f , and to $\mathbf{x}_j = f(t_j)$ as the switching points of f . Fixing a number $\ell \in \mathbb{N}$, we want to construct a PWL function γ_ℓ , consisting of m affine pieces p'_1, \dots, p'_m with ℓ different slopes, with the same switching times as in f , with switching points $\mathbf{y}_0, \dots, \mathbf{y}_m$ ε -close to those in f (which is necessary and sufficient for $d(f, \gamma_\ell) \leq \varepsilon$), and with unknown slopes $\mathbf{b}_1 = \text{slope}(p'_1), \dots, \mathbf{b}_m = \text{slope}(p'_m)$. We define the logical formula

$$\phi_{f,\varepsilon}(\ell) := \bigwedge_{j=1}^m \mathbf{y}_j = \mathbf{y}_{j-1} + \mathbf{b}_j(t_j - t_{j-1}) \wedge \bigwedge_{j=0}^m \mathbf{y}_j \in [\mathbf{x}_j]_\varepsilon \wedge \bigwedge_{j=1}^m \bigvee_{k=1}^\ell \mathbf{b}_j = \mathbf{c}_k,$$

which is satisfiable if and only if there exists a suitable PWL function γ_ℓ . For lifting to a set of functions \mathcal{F} , we define the formula $\phi_{\mathcal{F},\varepsilon}(\ell) := \bigwedge_{f \in \mathcal{F}} \phi_{f,\varepsilon}(\ell)$. These formulae fall into the theory of linear arithmetic and can be effectively solved by an SMT solver. Now, we can state the following results.

Lemma 1. *Let \mathcal{F} be a finite set of PWL functions and $\varepsilon \in \mathbb{R}_{\geq 0}$. If $\phi_{\mathcal{F},\varepsilon}(\ell)$ is satisfiable for some integer value ℓ , then there exists a set of PWL functions \mathcal{F}' such that $|\mathcal{F}'| = |\mathcal{F}|$, each function in \mathcal{F} is ε -close to some function in \mathcal{F}' , and the number of distinct slopes in \mathcal{F}' does not exceed ℓ .*

The set \mathcal{F}' can be extracted from a satisfying assignment. We define a hybrid automaton with minimal number of locations 0-capturing a given PWL function.

Definition 4 (Canonical automaton). Let f be an n -PWL function. The canonical automaton of f is $\mathcal{H}_f := (Q, E, \mathbb{R}^n, \text{Flow}, \text{Inv}, \text{Grd})$ with

- $Q = \{q_a \mid \exists p \in \text{pieces}(f) : \text{slope}(p) = a\}$,
- $E = \{(q_a, q_{a'}) \mid \exists p, p' \in \text{pieces}(f) \text{ adjacent} : \text{slope}(p) = a, \text{slope}(p') = a'\}$,
- $\text{Flow}(q_a) = a$,
- $\text{Inv}(q_a) = \text{chull}(\{\text{img}(p) \mid p \in \text{pieces}(f) : \text{slope}(p) = a\})$, and
- $\text{Grd}((q_a, q_{a'})) = \text{chull}(\{\text{end}(p) \mid \exists p, p' \in \text{pieces}(f) \text{ adjacent} : \text{slope}(p) = a, \text{slope}(p') = a'\})$.

Lemma 2. Given a PWL function f , the canonical automaton \mathcal{H}_f 0-captures f , and every LHA that 0-captures f has at least as many modes as \mathcal{H}_f .

Definition 5 (Merging). Given two hybrid automata $\mathcal{H}_i = (Q_i, E_i, X, \text{Flow}_i, \text{Inv}_i, \text{Grd}_i)$, $i = 1, 2$ with $Q_1 \cap Q_2 = \emptyset$, let $Q_a = Q_a^{\mathcal{H}_1} \cup Q_a^{\mathcal{H}_2}$ be the locations with flow equal to a . We define the merging of \mathcal{H}_1 and \mathcal{H}_2 as $\mathcal{H}_1 \sqcup \mathcal{H}_2 := (Q, E, X, \text{Flow}, \text{Inv}, \text{Grd})$ with $Q = \{q_a \mid a \in \mathbb{R}^n, Q_a \neq \emptyset\}$, $E = \{(q_a, q_{a'}) \mid \exists (q, q') \in E_1 \cup E_2, q \in Q_a, q' \in Q_{a'}\}$, $\text{Flow}(q_a) = a$, $\text{Inv}(q_a) = \text{chull}(\{\text{Inv}_i(q) \mid q \in Q_a, i = 1, 2\})$, and $\text{Grd}((q_a, q_{a'})) = \text{chull}(\{\text{Grd}_i((q, q')) \mid (q, q') \in E_i, q \in Q_a, q' \in Q_{a'}, i = 1, 2\})$.

Theorem 1. Given a finite set of PWL functions \mathcal{F} and a value $\varepsilon \in \mathbb{R}_{\geq 0}$, let ℓ be the smallest integer such that $\phi_{\mathcal{F}, \varepsilon}(\ell)$ is satisfiable and let \mathcal{F}' be a set of PWL functions corresponding to a satisfying assignment. Then, the merging of canonical automata $\sqcup_{f \in \mathcal{F}'} \mathcal{H}_f$ solves Problem 2.

The above synthesis algorithm works well with short and low-dimensional PWL functions but does not scale to realistic problem sizes due to the heavy use of disjunctions. We next address scalability with a new online algorithm.

3.2 Asynchronous Switching Specification

We now change the requirement from the previous subsection (minimality in the models' discrete structure) to tightness in the model's state-space constraints. Intuitively, for every vertex v of an invariant or guard in \mathcal{H} there should be some witness data $f \in \mathcal{F}$ that is close to v (at some point in time).

Definition 6 (Precision). Given an LHA $\mathcal{H} = (Q, E, X, \text{Flow}, \text{Inv}, \text{Grd})$, let $\text{vert}(\mathcal{H})$ denote the union of the vertices of the invariants and guards:

$$\text{vert}(\mathcal{H}) = \bigcup_{q \in Q} \text{vert}(\text{Inv}(q)) \cup \bigcup_{e \in E} \text{vert}(\text{Grd}(e))$$

Given a set of PWL functions \mathcal{F} and a value $\varepsilon \in \mathbb{R}_{>0}$, we say that \mathcal{H} is ε -precise (with respect to \mathcal{F}) if the following holds:

$$\forall v \in \text{vert}(\mathcal{H}) \exists f \in \mathcal{F} \exists t \in \text{dom}(f) : \|v - f(t)\| \leq \varepsilon.$$

The restriction to the vertices is reasonable because all sets are compact convex polyhedra. Note that ε -capturing compares functions to the automaton's executions, while ε -precision compares functions to the automaton's state-space.

We also relax the limitation to synchronously switching executions. Instead, we allow *asynchronous* switching, characterized as follows: for every function f ε -captured by \mathcal{H} , there exists an execution $\sigma \in \text{exec}(\mathcal{H})$ with the same number of switches as there are kinks in f , i.e., $\text{len}(\mathcal{I}) = |\text{pieces}(f)|$, and where the j -th switch in the execution should take place during the time period between the kinks $j - 1$ and $j + 1$. We close this section with the new problem statement (a refinement of Problem 1), and present a solution in the next section.

Problem 3 (Asynchronous synthesis). Given a finite set of PWL functions \mathcal{F} and a value $\varepsilon \in \mathbb{R}_{\geq 0}$, construct an ε -precise LHA \mathcal{H} that ε -captures every function $f \in \mathcal{F}$ asynchronously.

4 Membership-based Synthesis Approach

In this section, we present an algorithm for solving Problem 3. The core of the algorithm is a reachability computation for providing the polyhedral regions where executions of an LHA that are ε -close to a given PWL function f are allowed to switch. More precisely, given a path π and the ε -tube of f , the algorithm iteratively constructs the set inside the ε -tube where an execution following π can switch, without escaping from the tube. These reachable set are, in general, computed with respect to a starting compact convex polyhedron P , a pair of adjacent affine pieces p and p' , and a pair of modes q and q' along π .

Definition 7. Given an LHA $\mathcal{H} = (Q, E, X, \text{Flow}, \text{Inv}, \text{Grd})$ and a value $\varepsilon \in \mathbb{R}_{\geq 0}$, a reachable switching set $\text{switch}_{\mathcal{H}}(P, p, p', q, q')$ from a set P with respect to two adjacent affine pieces p, p' and a path $\pi := q, q'$ in \mathcal{H} is defined as

$$\{\mathbf{x} \in \text{Grd}((q, q')) \mid \exists \sigma = (\mathcal{I}, \gamma, \delta) \in \text{exec}(\mathcal{H}) : \sigma \xrightarrow{\mathcal{H}} \pi, \text{dom}(\gamma) = \text{dom}(p) \cup \text{dom}(p'), \gamma(0) \in P, \gamma(t) \in \text{tube}_{p, \varepsilon}(t) \cup \text{tube}_{p', \varepsilon}(t), \text{ and } \mathbf{x} = \gamma(\text{ub}(\mathcal{I}(0)))\}.$$

Inductive Reachable Switching Computation. Given an LHA \mathcal{H} , an m -PWL function $f = p_1, \dots, p_m$, a value $\varepsilon \in \mathbb{R}_{\geq 0}$ and a path $\pi = q_1, \dots, q_m$ in the graph $(Q_{\mathcal{H}}, E_{\mathcal{H}})$, we compute the reachable switching set P_j^{π} for every $0 \leq j \leq m$:

- $P_0^{\pi} := \text{Inv}_{\mathcal{H}}(q_1) \cap \text{tube}_{f, \varepsilon}(0)$,
- $P_j^{\pi} := \text{switch}_{\mathcal{H}}(P_{j-1}^{\pi}, p_{j-1}, p_j, q_{j-1}, q_j)$ for $1 < j < m$, and
- $P_m^{\pi} := \{\mathbf{x} \in \text{Inv}(q_m) \mid \exists \sigma = (\mathcal{I}, \gamma, \delta) \in \text{exec}(\mathcal{H}) : \sigma \xrightarrow{\mathcal{H}} q_m, \gamma(0) \in P_{m-1}^{\pi}, \text{dom}(\gamma) = \text{dom}(p_m), \gamma(t) \in \text{tube}_{p_m, \varepsilon}(t) \text{ and } \mathbf{x} = \gamma(\text{ub}(\mathcal{I}(m)))\}$.

We denote the set of all reachable switching sets P_j^{π} by \mathcal{P}^{π} . We are now ready to present the complete synthesis algorithm.

Algorithm 1. SYNTHESIS

Input: A set of PWL functions $\mathcal{F} = \{f_0, \dots, f_N\}$ and a value $\varepsilon \in \mathbb{R}_{\geq 0}$

Output: A linear hybrid automaton \mathcal{H} that solves Problem 3

```

1:  $\mathcal{H} := \text{INITLHA}(f_0, \varepsilon)$                                  $\triangleright$  construct initial model for  $\varepsilon$ -capturing  $f_0$ 
2: for  $f \in \mathcal{F} \setminus \{f_0\}$  do
3:    $(ans, \pi) := \text{MEMBERSHIP}(f, \mathcal{H}, \varepsilon)$ 
4:   if not  $ans$  then
5:      $\bar{\mathcal{H}} := \text{RELAXALL}(\mathcal{H}, f, \varepsilon)$                  $\triangleright$  relax model constraints entirely
6:      $(ans, \pi) := \text{MEMBERSHIP}(f, \bar{\mathcal{H}}, \varepsilon)$ 
7:   if  $ans$  then
8:      $\mathcal{H} := \text{RELAXPATH}(\mathcal{H}, f, \varepsilon, \pi)$      $\triangleright$  relax model constraints for  $\varepsilon$ -capturing  $f$ 
9:   else
10:     $\mathcal{H} := \text{ADAPT}(\mathcal{H}, f, \varepsilon, \pi)$             $\triangleright$  adapt model for  $\varepsilon$ -capturing  $f$ 
11: return  $\mathcal{H}$ 

```

4.1 Membership-based Synthesis Algorithm

The synthesis algorithm outlined in Algorithm 1 computes an LHA \mathcal{H} solving Problem 3 for a given finite set of PWL functions \mathcal{F} and a value $\varepsilon \in \mathbb{R}_{\geq 0}$. The algorithm initially infers an LHA \mathcal{H} that ε -captures the first function f_0 of \mathcal{F} in an ε -precise manner in line 1. The remaining PWL functions are handled in an iterative loop. For each PWL function f , the algorithm performs a membership query, where it checks if f is ε -captured by the LHA \mathcal{H} in line 3. If the query results in a positive answer ($ans = True$), nothing needs to be done. Otherwise, the query returns a path π and the LHA \mathcal{H} needs to be modified. The modification of the automaton \mathcal{H} is performed in two attempts. The first attempt, in line 5, temporarily increases invariants and guards of \mathcal{H} . If such a modification is sufficient to let the membership query succeed, the modifications are made permanent in line 8. Otherwise, in the second attempt the algorithm adds new modes and/or transitions to \mathcal{H} along the path π . Below we describe every procedure of Algorithm 1 in detail.

Initialization. The procedure $\text{INITLHA}(f, \varepsilon)$ constructs an initial LHA \mathcal{H} that ε -captures an m -PWL function f . Observe that by Lemma 2 the canonical automaton \mathcal{H}_f 0-captures (and hence ε -captures) the function f . In order to allow similar dynamical behaviors in a given LHA \mathcal{H} , the procedure $\text{INITLHA}(f, \varepsilon)$ ε -bloats both invariant and guards polyhedra. The procedure $\text{INITLHA}(f, \varepsilon)$ outputs the ε -bloated canonical automaton $\mathcal{H}_f^\varepsilon$ and is illustrated in Fig. 1.

Definition 8. Given an LHA $\mathcal{H} = (Q, E, X, \text{Flow}, \text{Inv}, \text{Grd})$, we define the ε -bloated LHA of \mathcal{H} as $\mathcal{H}^\varepsilon = (Q, E, X, \text{Flow}, \text{Inv}^\varepsilon, \text{Grd}^\varepsilon)$ where $\text{Inv}^\varepsilon(q) = \lceil \text{Inv}(q) \rceil_\varepsilon$ for every $q \in Q$ and $\text{Grd}^\varepsilon(e) = \lceil \text{Grd}(e) \rceil_\varepsilon$ for every $e \in E$.

Lemma 3. Given a PWL function f and $\varepsilon \in \mathbb{R}_{\geq 0}$, $\mathcal{H}_f^\varepsilon$ ε -captures f .

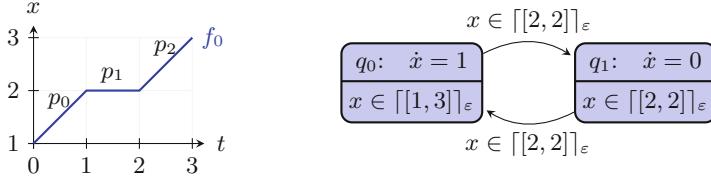


Fig. 1. Example describing the procedure $\text{INITLHA}(f, \varepsilon)$ for a 3-PWL function $f = f_0$ (depicted on the left). The function f_0 consists of three pieces p_0, p_1, p_2 with slopes 1, 0, 1, respectively. The LHA on the right is constructed as follows. Mode q_0 corresponds to pieces p_0 and p_2 ; the invariant is the ε -bloating of interval $[1, 3]$ (which is the convex hull of every start and end point in both pieces). Likewise, mode q_1 corresponds to piece p_1 . Transitions and their guards correspond to the kinks of f_0 at $t = 1$ and $t = 2$.

Membership. The procedure $\text{MEMBERSHIP}(f, \mathcal{H}, \varepsilon)$ checks whether there exists an *asynchronous* execution $\sigma = (\mathcal{I}, \gamma, \delta)$ in \mathcal{H} such that $d(f, \gamma) \leq \varepsilon$ holds. Let us introduce the required notions to formalize the membership problem.

Definition 9. An execution $\sigma = (\mathcal{I}, \gamma, \delta)$ of an LHA \mathcal{H} is consistent with an m -PWL function f , described by the affine pieces p_1, \dots, p_m , if $\text{len}(\mathcal{I}) = m$, $[\mathcal{I}] = \text{dom}(f)$, and $\text{ub}(\mathcal{I}(j)) \in \text{dom}(p_j) \cup \text{dom}(p_{j+1})$ for every $1 \leq j < m$.

Problem 4 (Membership). Given an m -PWL function f , an LHA \mathcal{H} , and a value $\varepsilon \in \mathbb{R}_{\geq 0}$, decide if there exists an execution $\sigma = (\mathcal{I}, \gamma, \delta)$ in $\text{exec}(\mathcal{H})$ that is consistent with f and such that $d(f, \gamma) \leq \varepsilon$ holds.

The procedure $\text{MEMBERSHIP}(f, \mathcal{H}, \varepsilon)$ solves Problem 4 by computing the reachable switching sets for every path π of length m in \mathcal{H} until finding a path π where every reachable switching set P_j^π for $0 \leq j \leq m$ is nonempty. Upon finding a path π satisfying the previous constraints, $\text{MEMBERSHIP}(f, \mathcal{H}, \varepsilon)$ returns *True* as answer, together with the path π . If there does not exist such a path π , it returns *False* as answer. We show an example in Fig. 2(a). We remark that, for a fixed path, Problem 4 is a timestamp-generation problem [2] with the restriction to time intervals for switching and the ε -tube as solution corridor.

Lemma 4. Let \mathcal{H} be an LHA and f be an m -PWL function. Then there exists a path π of length m in \mathcal{H} such that the final reachable switching set P_m^π is not empty if and only if there exists an execution σ in $\text{exec}(\mathcal{H})$ solving Problem 4.

Relaxation. If $\text{MEMBERSHIP}(f, \mathcal{H}, \varepsilon)$ returns *False*, $\text{RELAXALL}(\mathcal{H}, f, \varepsilon)$ constructs an automaton $\bar{\mathcal{H}}$ that is equivalent to \mathcal{H} except that its invariants and guards are enlarged to allow additional executions inside the $\text{tube}_{f, \varepsilon}$. Then, the algorithm computes $\text{MEMBERSHIP}(f, \bar{\mathcal{H}}, \varepsilon)$. If the answer is *False* again, the algorithm proceeds to the adaptation procedure in line 10. Otherwise (if the answer is *True*), we obtain a path π in $\bar{\mathcal{H}}$. Then the algorithm executes the procedure $\text{RELAXPATH}(\mathcal{H}, f, \varepsilon, \pi)$, which extends the constraints of invariants and guards

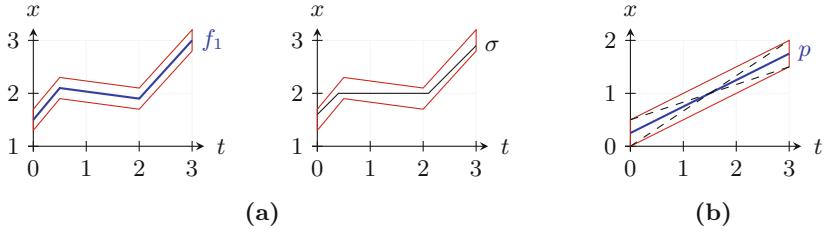


Fig. 2. (a) Example describing the procedure $\text{MEMBERSHIP}(f, \mathcal{H}, \varepsilon)$. On the left we depict a 3-PWL function f_1 and its ε -tube. On the right we show a possible execution in the LHA from Fig. 1. (b) Given an affine piece p , we say that another piece has a *similar* slope if it does not leave the tube. In the figure, we show the minimal and the maximal allowed slopes by dashed segments.

in \mathcal{H} for the modes in π by taking the convex hull with the corresponding reachable switching sets $P_j^\pi \in \mathcal{P}^\pi$. The relaxation procedure applied on the running example is shown in Fig. 3.

Adaptation. If both the membership query and the relaxation procedure fail, the procedure $\text{ADAPT}(\mathcal{H}, f, \varepsilon, \pi)$ modifies the LHA \mathcal{H} for ε -capturing f . Conceptually, we construct a new path π' , based on some path π , and modify \mathcal{H} accordingly such that the graph of \mathcal{H} contains π' . Recalling Lemma 4, we need to ensure that every reachable switching set in $\mathcal{P}^{\pi'}$ is nonempty. We construct π' by trying to preserve the modes in path π . If this is not possible, we try to replace them by existing modes in the LHA \mathcal{H} whenever possible, potentially adding new transitions. The last option is to create new modes. Finally, we extend the LHA \mathcal{H} by adding the new transitions and/or modes determined by the new path π' .

In more detail, given an LHA \mathcal{H} , an m -PWL function f and a path $\pi = q_1, \dots, q_m$ in \mathcal{H} , we start with path $\pi' = \pi$. Then, the adaptation procedure checks whether there is an empty reachable switching set in $\mathcal{P}^{\pi'}$. Every time we detect emptiness of the set $P_j^{\pi'}$ for some $0 \leq j \leq m$, a mode in the path π' is replaced in order to make $P_j^{\pi'}$ nonempty. We first try to replace the mode q_{j+1} if it exists. If $P_j^{\pi'}$ is still empty or q_{j+1} does not exist, we repeat the replacement for q_j , q_{i-1} , and so on, until $P_j^{\pi'}$ finally becomes nonempty.

For the replacement of the j -th mode q in the path π' we follow two strategies. The first strategy is to replace the mode q by an existing mode $q' \neq q$ in \mathcal{H} such that $\text{Flow}_{\mathcal{H}}(q')$ is *similar* to $\text{slope}(p_j)$. Formally, let T be the duration of piece p_j . $\text{Flow}_{\mathcal{H}}(q')$ is similar to $\text{slope}(p_j)$ if $\|\text{init}(p_j) + T \cdot \text{Flow}_{\mathcal{H}}(q') - \text{end}(p_j)\| \leq 2\varepsilon$. See Fig. 2(b) for an example. If the first strategy fails, the second strategy is to create a new mode q^* with flow $\text{newflow}(q^*) = \text{slope}(p_j)$ for replacement in π' . We denote the set of existing modes similar to some mode q in π by $\text{sim}(\pi')$, and the set of new modes q^* by $\text{new}(\pi')$. Once the path π' is constructed, the adaptation of the LHA \mathcal{H} is performed with respect to π' . Figure 4 exemplifies the adaptation of the LHA in Fig. 1.

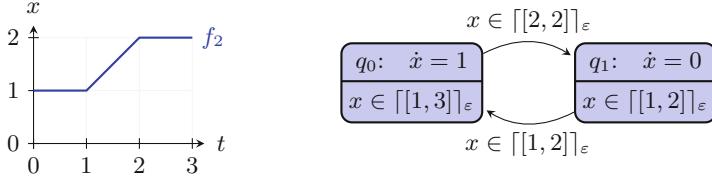


Fig. 3. Example describing the procedure $\text{RELAXPATH}(\mathcal{H}, f, \varepsilon, \pi)$ for \mathcal{H} given in Fig. 1, $f = f_2$ (depicted on the left), and path $\pi = q_1, q_0, q_1$. The algorithm increases the invariant of mode q_1 by computing the convex hull of the old invariant $[[2, 2]]_\varepsilon$ and the set $[[1, 1]]_\varepsilon$. Analogously, the guard of the transition (q_1, q_0) is increased.

Definition 10. The adaptation of the LHA $\mathcal{H} = (Q, E, X, \text{Flow}, \text{Inv}, \text{Grd})$ with respect to an m -PWL function f with affine pieces p_1, \dots, p_m and a path $\pi = q_1, \dots, q_m$ is the LHA $\mathcal{H}' = (Q', E', X, \text{Flow}', \text{Inv}', \text{Grd}')$ defined as:

- $Q' := Q \cup \text{new}(\pi'),$
- $E' := E \cup \{(q_j, q_{j+1}) \mid 1 \leq j < m\},$
- $\text{Flow}'(q) := \begin{cases} \text{newflow}(q) & \text{if } q \in \text{new}(\pi'), \\ \text{Flow}(q) & \text{otherwise,} \end{cases}$
- $\text{Inv}'(q) := \begin{cases} \text{chull}(\bigcup_{q=q_j, q \neq q_1} P_{j-1}^{\pi'} \cup \bigcup_{q=q_j} P_j^{\pi'}) & \text{if } q \in \text{new}(\pi'), \\ \text{chull}(\text{Inv}(q) \cup \bigcup_{q=q_j, q \neq q_1} P_{j-1}^{\pi'} \cup \bigcup_{q=q_j} P_j^{\pi'}) & \text{if } q \in \text{sim}(\pi'), \\ \text{Inv}(q) & \text{otherwise,} \end{cases}$
- $\text{Grd}'((q, q')) := \begin{cases} \text{chull}(\bigcup_{q=q_j, q'=q_{j+1}} P_j^{\pi'}) & \text{if } q \in \text{new}(\pi') \\ \text{or } q' \in \text{new}(\pi'), \\ \text{chull}(\text{Grd}((q, q')) \cup \bigcup_{q=q_j, q'=q_{j+1}} P_j^{\pi'}) & \text{if } q \in \text{sim}(\pi') \\ \text{or } q' \in \text{sim}(\pi'), \\ \text{Grd}((q, q')) & \text{otherwise.} \end{cases}$

If there is no path of length m in the graph of \mathcal{H} , we choose a shorter path π in \mathcal{H} of length m' for the adaptation procedure. Then, for every position $j \geq m'$, we define the reachable switching set P_j^π as an empty set and proceed as usual.

4.2 Discussion

The construction of the initial LHA (line 1 in Algorithm 1) can be modified to *clustering* pieces with *similar* slopes. This can help reducing the number of modes in the initial automaton, but does not guarantee that the first PWL function f_0 is ε -captured. To fix this, f_0 can be included in the loop of Algorithm 1.

Algorithm 1 follows a *local* repair strategy, based on a single PWL function. Thanks to this, the algorithm can be used in an online setting where new data arrives after the algorithm has started. However, the resulting model is influenced

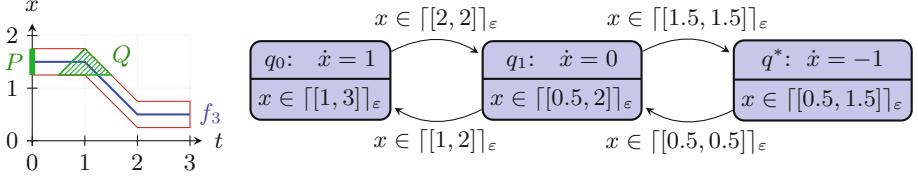


Fig. 4. Example describing the procedure $\text{ADAPT}(\mathcal{H}, f, \pi, \varepsilon)$ for the LHA \mathcal{H} in Fig. 1 with respect to the 3-PWL function $f = f_3$ and the path $\pi = q_1, q_0, q_1$ and $\varepsilon = 0.25$. The initial reachable switching set P_0^π is the projection of the set P on state x . Considering the flows in q_1 and q_0 , the next reachable switching set P_1^π is the projection of the set Q on state x . Observe that from Q , using the flow of q_1 , the reachable switching set P_2^π is empty. We thus add a new mode q^* and obtain the new path $\pi' = q_1, q^*, q_1$.

by the order in which the algorithm processes the functions $f \in \mathcal{F}$. In the simple case that \mathcal{F} only contains affine functions with the same slope, all models resulting from different processing orders will consist of a single mode with the same flow, and the invariant bounds differ by at most ε . Furthermore, for a precision value $\varepsilon = 0$, the result is always order-independent.

We now discuss the restrictions of the models we obtain from Algorithm 1. We did not include a set of initial states in our presentation, but the generalization is straightforward. Our transitions do not include assignments, which would make executions discontinuous. The usual assumption in many application domains, e.g., life sciences, is that the underlying system is continuous, so having assignments would not be desirable. In the setting where the input is given as time-series data, discrete events would typically be approximated by steep slopes in the PWL function. In the setting where the input is given as discontinuous PWL functions f , in order to ε -capture f , one would generally require that the automaton switches synchronously with f (cf. Sect. 3.1), instead of asynchronous switching as in our algorithm. Under this additional assumption, we can pose the procedures **MEMBERSHIP** and **RELAXPATH** as a single linear program (similar to formula $\phi_{f,\varepsilon}$). This linear program can also be used to identify assignments.

The continuous dynamics of our models are defined by constant differential equations. As mentioned before, this class generally suffices to approximate an arbitrary continuous function (by increasing the number of modes). An extension of our approach to use polyhedral differential *inclusions* (also called linear envelopes) is by merging modes of “similar” dynamics. This may, however, lead to the dilemma that several modes are equally similar.

4.3 Theoretical Properties of the Membership-based Synthesis

The following theorem asserts that Algorithm 1 solves Problem 3.

Theorem 2 (Soundness and precision). *Given a finite set of PWL functions \mathcal{F} and a value $\varepsilon \in \mathbb{R}_{\geq 0}$, let \mathcal{H} be an automaton resulting from $\text{SYNTHESIS}(\mathcal{F}, \varepsilon)$. Then \mathcal{H} both ε -captures all functions in \mathcal{F} and is ε -precise with respect to \mathcal{F} .*

Algorithm 1 satisfies a completeness property in the following sense. For every model \mathcal{H} from a certain class we can find a set \mathcal{F} of PWL functions and a value ε such that $\text{SYNTHESIS}(\mathcal{F}, \varepsilon)$ results in \mathcal{H} . Before we can characterize the class of models, we first need to introduce some terminology.

Definition 11. *Let $q \in Q$ be a mode with invariant $X = \text{Inv}(q)$ and flow $\text{Flow}(q)$. We call a continuous state $\mathbf{x}_2 \in X$ forward reachable in q if there is a continuous state $\mathbf{x}_1 \in X$ such that \mathbf{x}_2 is reachable from \mathbf{x}_1 by just letting time pass, i.e., $\exists t > 0 : \mathbf{x}_2 = \mathbf{x}_1 + \text{Flow}(q) \cdot t$. Analogously, we call state $\mathbf{x}_2 \in X$ backward reachable in q if there is a state $\mathbf{x}_1 \in X$ such that \mathbf{x}_2 is reachable from \mathbf{x}_1 . A continuous state is dead in q if it is neither forward reachable nor backward reachable in q .*

We characterize the class of automata $\mathcal{H} = (Q, E, X, \text{Flow}, \text{Inv}, \text{Grd})$ for which the algorithm is complete by considering the following assumptions: (1) no invariant contains a dead continuous state. Furthermore, if $e = (q_1, q_2)$ is a transition, then all continuous states in the guard $\text{Grd}(e)$ are forward reachable in q_1 and backward reachable in q_2 , and (2) no two modes have the same slope \square

Roughly speaking, Assumption (1) asserts that, after every switch, an execution can stay in the new mode for a positive amount of time.

Theorem 3 (Completeness). *Given an LHA \mathcal{H} satisfying Assumptions (1) and (2), there exist PWL functions \mathcal{F} such that $\text{SYNTHESIS}(\mathcal{F}, 0)$ results in \mathcal{H} .*

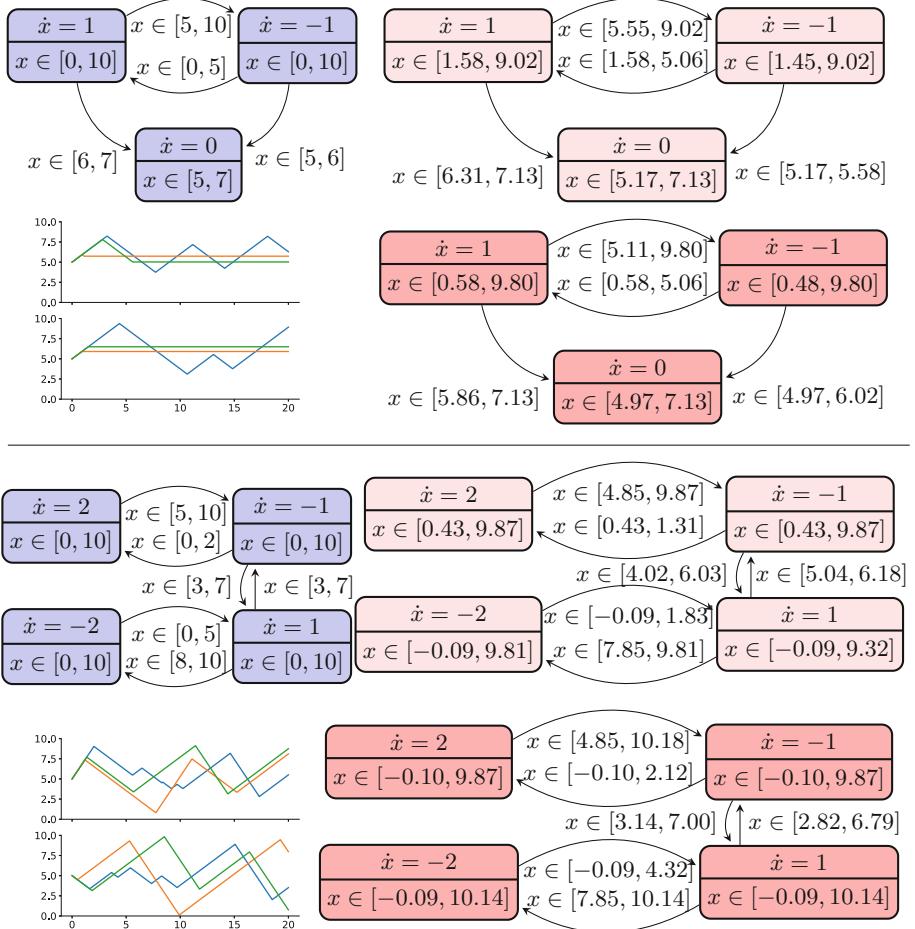
5 Experimental Results

In this section, we present the experiments used to evaluate our algorithm. The algorithm was implemented in Python and relies on the standard scientific computation packages. For the computations involving polyhedra we used the `pplpy` wrapper to the Parma Polyhedra Library [4].

Case Study: Online Synthesis. We evaluate the precision of our algorithm by collecting data from the executions of existing linear hybrid automata. For each given automaton, we randomly sample ten executions and pass them to our algorithm, which then constructs a new model. After that, we run our algorithm with another 90 executions, but we reuse the intermediate model, thus demonstrating the online feature of the algorithm. We show the different models for two hand-crafted examples in Table 1. We tried both sampling from random states and from a fixed state. The examples show the latter case, which makes sampling the complete state-space and thus learning a precise model harder.

The first example contains a sink with two incoming transitions, which requires at least two simulations to observe both transitions. Consequently, the algorithm had to make use of the *adaptation* step at least once to add one of the

Table 1. Synthesis results for two automaton models. The original model is shown in blue. The synthesis result after 10 iterations is shown in bright red, and after another 90 iterations in dark red. On the bottom left we show three sample executions starting from the same point (top: original model, bottom: synthesized model after 100 iterations). We used $\varepsilon = 0.2$ in all cases. Numbers are rounded to two places.



transitions. In the second example, some parts of the state-space are explored less frequently by the sampled executions. Hence the first model obtained after ten iterations does not represent all behavior of the original model yet. After the additional 90 iterations, the remaining parts of the state space have been visited, which is reflected in the precise bounds of the resulting model. In the table, we also show three sample executions from both the original and the final synthesized automaton to illustrate the similarity in the dynamical behavior.

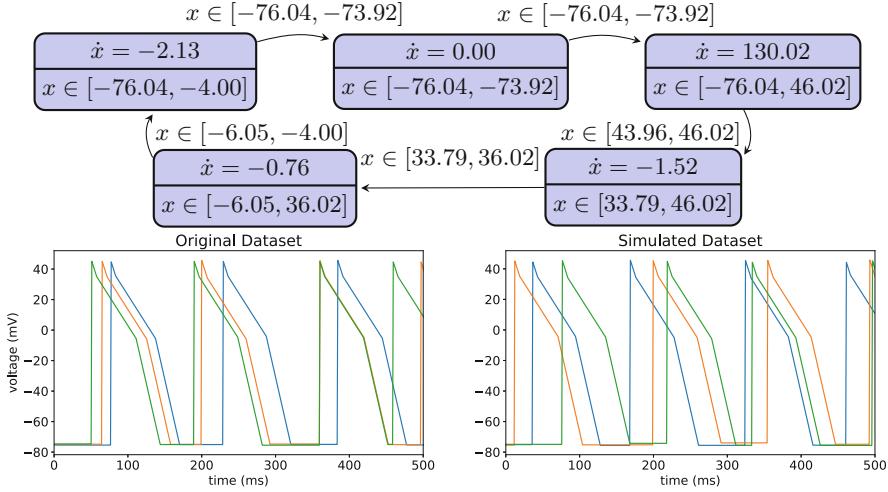


Fig. 5. Results for the cell model. Top: synthesized model using our algorithm. Bottom: three input traces (left) and random simulations of the synthesized model (right).

Case Study: Cell Model. For our case study we synthesize a hybrid automaton from voltage traces of excitable cells. Excitable cells are an important class of cells comprising neurons, cardiac cells, and other muscle cells. The main property of excitable cells is that they exhibit electrical activity which in the case of neurons enables signal transmission and in the case of muscle cells allows them to contract. The excitation signal usually follows distinct dynamics called action potential. Grosu et al. construct a *cyclic-linear hybrid automaton* from action-potential traces of cardiac cells [8]. In their model they identify six modes, two of which exhibit the same dynamics and are just used to model an input signal.

Our algorithm successfully synthesizes a model, depicted in Fig. 5, consisting of five modes that roughly match the normal phases of an action potential. We evaluate the quality of the synthesized model by simulating random executions and visually comparing to the original data (see the bottom of Fig. 5).

6 Conclusion

In this paper we have presented two fully automatic approaches to synthesize a linear hybrid automaton from data. As key features, the synthesized automaton captures the data up to a user-defined bound and is tight. Moreover, the online feature of the membership-based approach allows to combine the approach with alternative synthesis techniques, e.g., for constructing initial models.

A future line of work is to design a methodology for identification of weak generalizations in the model, and use them for driving the experiments and, in consequence, adjusting the model. We would first synthesize a model as before, but then identify the aspects of the model that are least substantiated by the

data (e.g., areas in the state space or specific sequences in the executions). Then we would query the system for data about those aspects, and repair the model accordingly. As another line of work, we plan to extend the approach to go from dynamics defined by piecewise-constant differential equations toward linear envelopes. Our approach can be seen as a generalization, to LHA, of Angluin's algorithm for constructing a finite-state machine from finite traces [3], and we plan to pursue this connection further.

References

1. 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.) HS 1991-1992. LNCS, vol. 736, pp. 209–229. Springer, Heidelberg (1993). https://doi.org/10.1007/3-540-57318-6_30
2. Alur, R., Kurshan, R.P., Viswanathan, M.: Membership questions for timed and hybrid automata. In: RTSS, pp. 254–263. IEEE Computer Society (1998). <https://doi.org/10.1109/REAL.1998.739751>
3. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. **75**(2), 87–106 (1987). [https://doi.org/10.1016/0890-5401\(87\)90052-6](https://doi.org/10.1016/0890-5401(87)90052-6)
4. 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). <https://doi.org/10.1016/j.scico.2007.08.001>
5. Bemporad, A., Garulli, A., Paoletti, S., Vicino, A.: A bounded-error approach to piecewise affine system identification. IEEE Trans. Autom. Control **50**(10), 1567–1580 (2005). <https://doi.org/10.1109/TAC.2005.856667>
6. Douglas, D.H., Peucker, T.K.: Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. Cartographica **10**(2), 112–122 (1973)
7. Garulli, A., Paoletti, S., Vicino, A.: A survey on switched and piecewise affine system identification. IFAC Proc. Vol. **45**(16), 344–355 (2012). <https://doi.org/10.3182/20120711-3-BE-2027.00332>
8. Grosu, R., Mitra, S., Ye, P., Entcheva, E., Ramakrishnan, I.V., Smolka, S.A.: Learning cycle-linear hybrid automata for excitable cells. In: Bemporad, A., Bicchi, A., Buttazzo, G. (eds.) HSCC 2007. LNCS, vol. 4416, pp. 245–258. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71493-4_21
9. Hakimi, S.L., Schmeichel, E.F.: Fitting polygonal functions to a set of points in the plane. CVGIP Graph. Model. Image Process. **53**(2), 132–136 (1991). [https://doi.org/10.1016/1049-9652\(91\)90056-P](https://doi.org/10.1016/1049-9652(91)90056-P)
10. Hashambhoy, Y., Vidal, R.: Recursive identification of switched ARX models with unknown number of models and unknown orders. In: CDC, pp. 6115–6121 (2005). <https://doi.org/10.1109/CDC.2005.1583140>
11. 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 (Series F: Computer and Systems Sciences), vol. 170, pp. 265–292. Springer, Berlin, Heidelberg (2000). https://doi.org/10.1007/978-3-642-59615-5_13

12. Lamrani, I., Banerjee, A., Gupta, S.K.S.: HyMn: mining linear hybrid automata from input output traces of cyber-physical systems. In: ICPS, pp. 264–269. IEEE (2018). <https://doi.org/10.1109/ICPHYS.2018.8387670>
13. Liberzon, D.: Switching in Systems and Control. Birkhäuser, Boston (2003). <https://doi.org/10.1007/978-1-4612-0017-8>
14. Ly, D.L., Lipson, H.: Learning symbolic representations of hybrid dynamical systems. JMLR **13**, 3585–3618 (2012). <http://dl.acm.org/citation.cfm?id=2503356>
15. Medhat, R., Ramesh, S., Bonakdarpour, B., Fischmeister, S.: A framework for mining hybrid automata from input/output traces. In: EMSOFT, pp. 177–186. IEEE (2015). <https://doi.org/10.1109/EMSOFT.2015.7318273>
16. Niggemann, O., Stein, B., Vodencarevic, A., Maier, A., Kleine Büning, H.: Learning behavior models for hybrid timed systems. In: AAAI. AAAI Press (2012). <http://www.aaai.org/ocs/index.php/AAAI/AAAI12/paper/view/4993>
17. Ozay, N.: An exact and efficient algorithm for segmentation of ARX models. In: ACC, pp. 38–41. IEEE (2016). <https://doi.org/10.1109/ACC.2016.7524888>
18. Paoletti, S., Juloski, A.L., Ferrari-Trecate, G., Vidal, R.: Identification of hybrid systems: a tutorial. Eur. J. Control **13**(2–3), 242–260 (2007). <https://doi.org/10.3166/ejc.13.242-260>
19. Skeppstedt, A., Lennart, L., Millnert, M.: Construction of composite models from observed data. Int. J. Control **55**(1), 141–152 (1992). <https://doi.org/10.1080/00207179208934230>
20. Summerville, A., Osborn, J.C., Mateas, M.: CHARDA: causal hybrid automata recovery via dynamic analysis. In: IJCAI, pp. 2800–2806. ijcai.org (2017). <https://doi.org/10.24963/ijcai.2017/390>
21. Verwer, S.: Efficient identification of timed automata: theory and practice. Ph.D. thesis, Delft University of Technology, Netherlands (2010). <http://resolver.tudelft.nl/uuid:61d9f199-7b01-45be-a6ed-04498113a212>
22. Vidal, R., Anderson, B.D.O.: Recursive identification of switched ARX hybrid models: exponential convergence and persistence of excitation. In: CDC, vol. 1, pp. 32–37 (2004). <https://doi.org/10.1109/CDC.2004.1428602>

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Overfitting in Synthesis: Theory and Practice

Saswat Padhi^{1(✉)}, Todd Millstein¹, Aditya Nori², and Rahul Sharma³

¹ University of California, Los Angeles, USA

{padhi,todd}@cs.ucla.edu

² Microsoft Research, Cambridge, UK

adityan@microsoft.com

³ Microsoft Research, Bengaluru, India

rahsha@microsoft.com

Abstract. In syntax-guided synthesis (SyGuS), a synthesizer’s goal is to automatically generate a program belonging to a grammar of possible implementations that meets a logical specification. We investigate a common limitation across state-of-the-art SyGuS tools that perform counterexample-guided inductive synthesis (CEGIS). We empirically observe that as the expressiveness of the provided grammar increases, the performance of these tools degrades significantly.

We claim that this degradation is not only due to a larger search space, but also due to *overfitting*. We formally define this phenomenon and prove *no-free-lunch* theorems for SyGuS, which reveal a fundamental tradeoff between synthesizer performance and grammar expressiveness.

A standard approach to mitigate overfitting in machine learning is to run multiple learners with varying expressiveness in parallel. We demonstrate that this insight can immediately benefit existing SyGuS tools. We also propose a novel single-threaded technique called *hybrid enumeration* that interleaves different grammars and outperforms the winner of the 2018 SyGuS competition (Inv track), solving more problems and achieving a 5× mean speedup.

1 Introduction

The *syntax-guided synthesis* (SyGuS) framework [3] provides a unified format to describe a program synthesis problem by supplying (1) a logical specification for the desired functionality, and (2) a grammar of allowed implementations. Given these two inputs, a SyGuS tool searches through the programs that are permitted by the grammar to generate one that meets the specification. Today, SyGuS is at the core of several state-of-the-art program synthesizers [5, 14, 23, 24, 29], many of which compete annually in the SyGuS competition [1, 4].

We demonstrate empirically that five state-of-the-art SyGuS tools are very sensitive to the choice of grammar. Increasing grammar expressiveness allows the tools to solve some problems that are unsolvable with less-expressive grammars. However, it also causes them to fail on many problems that the tools are able to solve with a less expressive grammar. We analyze the latter behavior both

S. Padhi—Contributed during an internship at Microsoft Research, India.

© The Author(s) 2019

I. Dillig and S. Tasiran (Eds.): CAV 2019, LNCS 11561, pp. 315–334, 2019.

https://doi.org/10.1007/978-3-030-25540-4_17

theoretically and empirically and present techniques that make existing tools much more robust in the face of increasing grammar expressiveness.

We restrict our investigation to a widely used approach [6] to SyGuS called *counterexample-guided inductive synthesis* (CEGIS) [37, §5]. In this approach, the synthesizer is composed of a learner and an oracle. The learner iteratively identifies a candidate program that is consistent with a given set of examples (initially empty) and queries the oracle to either prove that the program is *correct*, i.e., meets the given specification, or obtain a counterexample that demonstrates that the program does not meet the specification. The counterexample is added to the set of examples for the next iteration. The iterations continue until a correct program is found or resource/time budgets are exhausted.

Overfitting. To better understand the observed performance degradation, we instrumented one of these SyGuS tools (Sect. 2.2). We empirically observe that for a large number of problems, the performance degradation on increasing grammar expressiveness is often accompanied by a significant increase in the number of counterexamples required. Intuitively, as grammar expressiveness increases so does the number of *spurious* candidate programs, which satisfy a given set of examples but violate the specification. If the learner picks such a candidate, then the oracle generates a counterexample, the learner searches again, and so on.

In other words, increasing grammar expressiveness increases the chances for *overfitting*, a well-known phenomenon in machine learning (ML). Overfitting occurs when a learned function explains a given set of observations but does not generalize correctly beyond it. Since SyGuS is indeed a form of function learning, it is perhaps not surprising that it is prone to overfitting. However, we identify its specific source in the context of SyGuS—the spurious candidates induced by increasing grammar expressiveness—and show that it is a significant problem in practice. We formally define the *potential for overfitting* (Ω), in Definition 7, which captures the number of spurious candidates.

No Free Lunch. In the ML community, this tradeoff between expressiveness and overfitting has been formalized for various settings as *no-free-lunch* (NFL) theorems [34, §5.1]. Intuitively such a theorem says that for every learner there exists a function that cannot be efficiently learned, where efficiency is defined by the number of examples required. We have proven corresponding NFL theorems for the CEGIS-based SyGuS setting (Theorems 1 and 2).

A key difference between the ML and SyGuS settings is the notion of *m-learnability*. In the ML setting, the learned function may differ from the true function, as long as this difference (expressed as an error probability) is relatively small. However, because the learner is allowed to make errors, it is in turn required to learn given an arbitrary set of m examples (drawn from some distribution). In contrast, the SyGuS learning setting is *all-or-nothing*—either the tool synthesizes a program that meets the given specification or it fails. Therefore, it would be overly strong to require the learner to handle an arbitrary set of examples.

Instead, we define a much weaker notion of m -learnability for SyGuS, which only requires that there *exist* a set of m examples for which the learner succeeds. Yet, our NFL theorem shows that even this weak notion of learnability can always be thwarted: given an integer $m \geq 0$ and an expressive enough (as a function of m) grammar, for every learner there exists a SyGuS problem that cannot be learned without access to more than m examples. We also prove that overfitting is inevitable with an expressive enough grammar (Theorems 3 and 4) and that the potential for overfitting increases with grammar expressiveness (Theorem 5).

Mitigating Overfitting. Inspired by *ensemble methods* [13] in ML, which aggregate results from multiple learners to combat overfitting (and underfitting), we propose PLEARN—a black-box framework that runs multiple parallel instances of a SyGuS tool with different grammars. Although prior SyGuS tools run multiple instances of learners with different random seeds [7, 20], to our knowledge, this is the first proposal to explore multiple grammars as a means to improve the performance of SyGuS. Our experiments indicate that PLEARN significantly improves the performance of five state-of-the-art SyGuS tools—CVC4 [7, 33], EUSOLVER [5], LOOPINVGEN [29], SKETCHAC [20, 37], and STOCH [3, III F].

However, running parallel instances of a synthesizer is computationally expensive. Hence, we also devise a white-box approach, called *hybrid enumeration*, that extends the enumerative synthesis technique [2] to efficiently interleave exploration of multiple grammars in a single SyGuS instance. We implement hybrid enumeration within LOOPINVGEN¹ and show that the resulting single-threaded learner, LOOPINVGEN+HE, has negligible overhead but achieves performance comparable to that of PLEARN for LOOPINVGEN. Moreover, LOOPINVGEN+HE significantly outperforms the winner [28] of the invariant-synthesis (Inv) track of 2018 SyGuS competition [4]—a variant of LOOPINVGEN specifically tuned for the competition—including a 5× mean speedup and solving two SyGuS problems that no tool in the competition could solve.

Contributions. In summary, we present the following contributions:

- (Section 2) We empirically observe that, in many cases, increasing grammar expressiveness degrades performance of existing SyGuS tools due to *overfitting*.
- (Section 3) We formally define overfitting and prove *no-free-lunch* theorems for the SyGuS setting, which indicate that overfitting with increasing grammar expressiveness is a fundamental characteristic of SyGuS.
- (Section 4) We propose two mitigation strategies – **(1)** a black-box technique that runs multiple parallel instances of a synthesizer, each with a different grammar, and **(2)** a single-threaded enumerative technique, called *hybrid enumeration*, that interleaves exploration of multiple grammars.
- (Section 5) We show that incorporating these mitigating measures in existing tools significantly improves their performance.

¹ Our implementation is available at <https://github.com/SaswatPadhi/LoopInvGen>.

2 Motivation

In this section, we first present empirical evidence that existing SyGuS tools are sensitive to changes in grammar expressiveness. Specifically, we demonstrate that as we increase the expressiveness of the provided grammar, every tool starts failing on some benchmarks that it was able to solve with less-expressive grammars. We then investigate one of these tools in detail.

2.1 Grammar Sensitivity of SyGuS Tools

We evaluated 5 state-of-the-art SyGuS tools that use very different techniques:

- SKETCHAC [20] extends the SKETCH synthesis system [37] by combining both explicit and symbolic search techniques.
- STOCH [3, III F] performs a stochastic search for solutions.
- EUSOLVER [5] combines enumeration with unification strategies.
- Reynolds et al. [33] extend CVC4 [7] with a refutation-based approach.
- LOOPINVGEN [29] combines enumeration and Boolean function learning.

We ran these five tools on 180 invariant-synthesis benchmarks, which we describe in Sect. 5. We ran the benchmarks with each of the six grammars of quantifier-free predicates, which are shown in Fig. 1. These grammars correspond to widely used abstract domains in the analysis of integer-manipulating programs—Equalities, Intervals [11], Octagons [25], Polyhedra [12], algebraic expressions (Polynomials) and arbitrary integer arithmetic (Peano) [30]. The $*_s$ operator denotes scalar multiplication, e.g., $(*_s 2 x)$, and $*_n$ denotes nonlinear multiplication, e.g., $(*_n x y)$.

In Fig. 2, we report our findings on running each benchmark on each tool with each grammar, with a 30-minute wall-clock timeout. For each $\langle \text{tool}, \text{grammar} \rangle$ pair, the y -axis shows

the number of failing benchmarks that the same tool is able to solve with a less-expressive grammar. We observe that, for each tool, the number of such failures increases with the grammar expressiveness. For instance, introducing the scalar multiplication operator $(*_s)$ causes CVC4 to fail on 21 benchmarks that it is able to solve with Equalities (4/21), Intervals (18/21), or Octagons (10/21). Similarly, adding nonlinear multiplication causes LOOPINVGEN to fail on 10 benchmarks that it can solve with a less-expressive grammar.

```

(b)  $\models \text{true} \mid \text{false} \mid \langle \text{Bool variables} \rangle$ 
      |  $(\text{not } b)$  |  $(\text{or } b \ b)$  |  $(\text{and } b \ b)$ 
(i)  $\models \langle \text{Int constants} \rangle \mid \langle \text{Int variables} \rangle$ 

▶ Additional rule in Equalities grammar :
(b)  $\models (= i \ i)$ 

▶ Additional rules in Intervals grammar :
(b)  $\models (> i \ i) \mid (>= i \ i)$ 
      |  $(< i \ i) \mid (<= i \ i)$ 

▶ Additional rules in Octagons grammar :
(i)  $\models (+ i \ i) \mid (- i \ i)$ 

▶ Additional rule in Polyhedra grammar :
(i)  $\models (*_s i \ i)$ 

▶ Additional rule in Polynomials grammar :
(i)  $\models (*_N i \ i)$ 

▶ Additional rule in Peano grammar :
(i)  $\models (\text{div } i \ i) \mid (\text{mod } i \ i)$ 
```

Fig. 1. Grammars of quantifier-free predicates over integers (We use the \models operator to append new rules to previously defined nonterminals.)

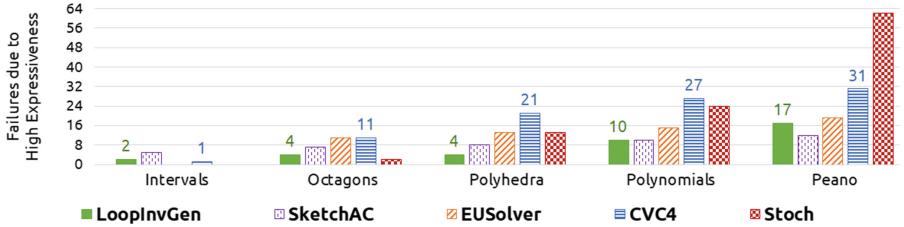


Fig. 2. For each grammar, each tool, the ordinate shows the number of benchmarks that fail with the grammar but are solvable with a less-expressive grammar.

	Increase (\uparrow)	Unchanged (=)	Decrease (\downarrow)
Expressiveness $\uparrow \wedge$ Time $\uparrow \rightarrow$ Rounds ?	27 %	67 %	6 %
Expressiveness $\uparrow \wedge$ Rounds $\uparrow \rightarrow$ Time ?	79 %	6 %	15 %

Fig. 3. Observed correlation between synthesis time and number of rounds, upon increasing grammar expressiveness, with LOOPINVGEN [29] on 180 benchmarks

2.2 Evidence for Overfitting

To better understand this phenomenon, we instrumented LOOPINVGEN [29] to record the candidate expressions that it synthesizes and the number of CEGIS iterations (called *rounds* henceforth). We compare each pair of successful runs of each of our 180 benchmarks on distinct grammars.² In 65 % of such pairs, we observe performance degradation with the more expressive grammar. We also report the correlation between performance degradation and number of rounds for the more expressive grammar in each pair in Fig. 3.

In 67 % of the cases with degraded performance upon increased grammar expressiveness, the number of rounds remains unaffected—indicating that this slowdown is mainly due to a larger search space. However, there is significant evidence of performance degradation due to *overfitting* as well. We note an increase in the number of rounds for 27 % of the cases with degraded performance. Moreover, we notice performance degradation in 79 % of all cases that required more rounds on increasing grammar expressiveness.

Thus, a more expressive grammar not only increases the search space, but also makes it more likely for LOOPINVGEN to overfit—select a spurious expression, which the oracle rejects with a counterexample, hence requiring more rounds. In the remainder of this section, we demonstrate this overfitting phenomenon on the verification problem shown in Fig. 4, an example by Gulwani and Jovic [17], which is the `fib_19` benchmark in the Inv track of SyGuS-Comp 2018 [4].

² We ignore failing runs since they require an unknown number of rounds.

For Fig. 4, we require an inductive invariant that is strong enough to prove that the assertion on line 6 always holds. In the SyGuS setting, we need to synthesize a predicate $\mathcal{I}: \mathbb{Z}^4 \rightarrow \mathbb{B}$ defined on a symbolic state $\sigma = \langle m, n, x, y \rangle$, that satisfies $\forall \sigma: \varphi(\mathcal{I}, \sigma)$ for the specification φ :³

$$\begin{aligned} \varphi(\mathcal{I}, \sigma) &\stackrel{\text{def}}{=} (0 \leq n \wedge 0 \leq m \leq n \wedge x = 0 \wedge y = m) \Rightarrow \mathcal{I}(\sigma) && \text{(precondition)} \\ &\wedge \forall \sigma': (\mathcal{I}(\sigma) \wedge T(\sigma, \sigma')) \Rightarrow \mathcal{I}(\sigma') && \text{(inductiveness)} \\ &\wedge (x \geq n \wedge \mathcal{I}(\sigma)) \Rightarrow y = n && \text{(postcondition)} \end{aligned}$$

where $\sigma' = \langle m', n', x', y' \rangle$ denotes the new state after one iteration, and T is a transition relation that describes the loop body:

$$\begin{aligned} T(\sigma, \sigma') &\stackrel{\text{def}}{=} (x < n) \wedge (x' = x + 1) \wedge (m' = m) \wedge (n' = n) \\ &\wedge [(x' \leq m \wedge y' = y) \vee (x' > m \wedge y' = y + 1)] \end{aligned}$$

Fig. 4. The fib_19 benchmark [17]

```

1 assume (0 ≤ n ∧ 0 ≤ m ≤ n)
2 assume (x = 0 ∧ y = m)
3 while (x < n) do
4     x ← x + 1
5     if (x > m) then y ← y + 1
6 assert (y = n)

```

Increasing expressiveness →					
Equalities	Intervals	Octagons	Polyhedra	Polynomials	Peano
× FAIL	0.32 s (19 rounds)	2.49 s (57 rounds)	2.48 s (57 rounds)	55.3 s (76 rounds)	68.0 s (88 rounds)

(a) Synthesis time and number of CEGIS iterations (rounds) with various grammars

- 16:** $(x \geq n) \vee (x + 1 < n) \vee (m \geq x \wedge m = y)$ **16:** $(x \geq n) \vee (x + 1 < n) \vee$
 $(2y = n) \vee (y(m - 1) = m)$
- 28:** $(x = y) \vee (y + m - n = x) \vee (x + 2 < n)$ **28:** $(y = 1) \vee (y = 0) \vee (m < 1) \vee (x^2 y > 1)$
- 57:** $(m = y) \vee (x \geq m \wedge x \geq y)$ **57:** $(x + 1 \geq n) \vee (x + 2 < n) \vee$
 $((m - n)(x - y) = 1)$

(b) Sample predicates with Polyhedra

Solution in both grammars: $(n \geq y) \wedge (y \geq x) \wedge ((m = y) \vee (x \geq m \wedge x \geq y))$

(c) Sample predicates with Peano

Fig. 5. Performance of LOOPIN VGEN [29] on the fib_19 benchmark (Fig. 4). In (b) and (c), we show predicates generated at various rounds (numbered in bold).

In Fig. 5(a), we report the performance of LOOPIN VGEN on fib_19 (Fig. 4) with our six grammars (Fig. 1). It succeeds with all but the least-expressive grammar. However, as grammar expressiveness increases, the number of rounds increase significantly—from 19 rounds with Intervals to 88 rounds with Peano.

LOOPIN VGEN converges to the *exact same* invariant with both Polyhedra and Peano but requires 30 more rounds in the latter case. In Figs. 5(b) and (c), we list some expressions synthesized with Polyhedra and Peano respectively. These expressions are solutions to intermediate subproblems—the final loop invariant is a conjunction of a subset of these expressions [29, §3.2]. Observe that the expressions generated with the Peano grammar are quite complex and unlikely to generalize well. Peano’s extra expressiveness leads to more spurious candidates, increasing the chances of overfitting and making the benchmark harder to solve.

³ We use \mathbb{B} , \mathbb{N} , and \mathbb{Z} to denote the sets of all Boolean values, all natural numbers (positive integers), and all integers respectively.

3 SyGuS Overfitting in Theory

In this section, first we formalize the *counterexample-guided inductive synthesis* (CEGIS) approach [37] to SyGuS, in which examples are iteratively provided by a verification oracle. We then state and prove *no-free-lunch* theorems, which show that there can be no optimal learner for this learning scheme. Finally, we formalize a natural notion of *overfitting* for SyGuS and prove that the potential for overfitting increases with grammar expressiveness.

3.1 Preliminaries

We borrow the formal definition of a SyGuS problem from prior work [3]:

Definition 1 (SyGuS Problem). *Given a background theory \mathbb{T} , a function symbol $f: X \rightarrow Y$, and constraints on f : (1) a semantic constraint, also called a specification, $\phi(f, x)$ over the vocabulary of \mathbb{T} along with f and a symbolic input x , and (2) a syntactic constraint, also called a grammar, given by a (possibly infinite) set \mathcal{E} of expressions over the vocabulary of the theory \mathbb{T} ; find an expression $e \in \mathcal{E}$ such that the formula $\forall x \in X: \phi(e, x)$ is valid modulo \mathbb{T} .*

We denote this SyGuS problem as $\langle f_{X \rightarrow Y} | \phi, \mathcal{E} \rangle_{\mathbb{T}}$ and say that it is satisfiable iff there exists such an expression e , i.e., $\exists e \in \mathcal{E}: \forall x \in X: \phi(e, x)$. We call e a satisfying expression for this problem, denoted as $e \models \langle f_{X \rightarrow Y} | \phi, \mathcal{E} \rangle_{\mathbb{T}}$.

Recall, we focus on a common class of SyGuS learners, namely those that learn from examples. First we define the notion of input-output (IO) examples that are consistent with a SyGuS specification:

Definition 2 (Input-Output Example). *Given a specification ϕ defined on $f: X \rightarrow Y$ over a background theory \mathbb{T} , we call a pair $\langle x, y \rangle \in X \times Y$ an input-output (IO) example for ϕ , denoted as $\langle x, y \rangle \models_{\mathbb{T}} \phi$ iff it is satisfied by some valid interpretation of f within \mathbb{T} , i.e.,*

$$\langle x, y \rangle \models_{\mathbb{T}} \phi \stackrel{\text{def}}{=} \exists e_* \in \mathbb{T}: e_*(x) = y \wedge (\forall x \in X: \phi(e_*, x))$$

The next two definitions respectively formalize the two key components of a CEGIS-based SyGuS tool: the verification oracle and the learner.

Definition 3 (Verification Oracle). *Given a specification ϕ defined on a function $f: X \rightarrow Y$ over theory \mathbb{T} , a verification oracle \mathcal{O}_{ϕ} is a partial function that given an expression e , either returns \perp indicating $\forall x \in X: \phi(e, x)$ holds, or gives a counterexample $\langle x, y \rangle$ against e , denoted as $e \rightsquigarrow_{\phi} \langle x, y \rangle$, such that*

$$e \rightsquigarrow_{\phi} \langle x, y \rangle \stackrel{\text{def}}{=} \neg \phi(e, x) \wedge e(x) \neq y \wedge \langle x, y \rangle \models_{\mathbb{T}} \phi$$

We omit ϕ from the notations \mathcal{O}_{ϕ} and \rightsquigarrow_{ϕ} when it is clear from the context.

Definition 4 (CEGIS-based Learner). A CEGIS-based learner $\mathcal{L}^{\mathcal{O}}(q, \mathcal{E})$ is a partial function that given an integer $q \geq 0$, a set \mathcal{E} of expressions, and access to an oracle \mathcal{O} for a specification ϕ defined on $f: X \rightarrow Y$, queries \mathcal{O} at most q times and either fails with \perp or generates an expression $e \in \mathcal{E}$. The trace

$$[e_0 \rightsquigarrow \langle x_0, y_0 \rangle, \dots, e_{p-1} \rightsquigarrow \langle x_{p-1}, y_{p-1} \rangle, e_p] \quad \text{where } 0 \leq p \leq q$$

summarizes the interaction between the oracle and the learner. Each e_i denotes the i^{th} candidate for f and $\langle x_i, y_i \rangle$ is a counterexample e_i , i.e.,

$$(\forall j < i: e_i(x_j) = y_j \wedge \phi(e_i, x_j)) \wedge (e_i \rightsquigarrow_{\phi} \langle x_i, y_i \rangle)$$

Note that we have defined oracles and learners as (partial) functions, and hence as *deterministic*. In practice, many SyGuS tools are deterministic and this assumption simplifies the subsequent theorems. However, we expect that these theorems can be appropriately generalized to randomized oracles and learners.

3.2 Learnability and No Free Lunch

In the machine learning (ML) community, the limits of learning have been formalized for various settings as *no-free-lunch* theorems [34, §5.1]. Here, we provide a natural form of such theorems for CEGIS-based SyGuS learning.

In SyGuS, the learned function must conform to the given grammar, which may not be fully expressive. Therefore we first formalize grammar expressiveness:

Definition 5 (k -Expressiveness). Given a domain X and range Y , a grammar \mathcal{E} is said to be k -expressive iff \mathcal{E} can express exactly k distinct $X \rightarrow Y$ functions.

A key difference from the ML setting is our notion of *m-learnability*, which formalizes the number of examples that a learner requires in order to learn a desired function. In the ML setting, a function is considered to *m-learnable* by a learner if it can be learned using an *arbitrary* set of m i.i.d. examples (drawn from some distribution). This makes sense in the ML setting since the learned function is allowed to make errors (up to some given bound on the error probability), but it is much too strong for the *all-or-nothing* SyGuS setting.

Instead, we define a much weaker notion of *m-learnability* for CEGIS-based SyGuS, which only requires that there *exist* a set of m examples that allows the learner to succeed. The following definition formalizes this notion.

Definition 6 (CEGIS-based m -Learnability). Given a SyGuS problem $S = \langle f_{X \rightarrow Y} \mid \phi, \mathcal{E} \rangle_{\mathbb{T}}$ and an integer $m \geq 0$, we say that S is m -learnable by a CEGIS-based learner \mathcal{L} iff there exists a verification oracle \mathcal{O} under which \mathcal{L} can learn a satisfying expression for S with at most m queries to \mathcal{O} , i.e., $\exists \mathcal{O}: \mathcal{L}^{\mathcal{O}}(m, \mathcal{E}) \models S$.

Finally we state and prove the no-free-lunch (NFL) theorems, which make explicit the tradeoff between grammar expressiveness and learnability. Intuitively, given an integer m and an expressive enough (as a function of m) grammar, for every learner there exists a SyGuS problem that cannot be solved without access to at least $m + 1$ examples. This is true despite our weak notion of learnability.

Put another way, as grammar expressiveness increases, so does the number of examples required for learning. On one extreme, if the given grammar is 1-expressive, i.e., can express exactly one function, then all satisfiable SyGuS problems are 0-learnable—no examples are needed because there is only one function to learn—but there are *many* SyGuS problems that cannot be satisfied by this function. On the other extreme, if the grammar is $|Y|^{|X|}$ -expressive, i.e., can express all functions from X to Y , then for every learner there exists a SyGuS problem that requires *all* $|X|$ examples in order to be solved.

Below we first present the NFL theorem for the case when the domain X and range Y are finite. We then generalize to the case when these sets may be countably infinite. We provide the proofs of these theorems in the extended version of this paper [27, Appendix A.1].

Theorem 1 (NFL in CEGIS-based SyGuS on Finite Sets). *Let X and Y be two arbitrary finite sets, \mathbb{T} be a theory that supports equality, \mathcal{E} be a grammar over \mathbb{T} , and m be an integer such that $0 \leq m < |X|$. Then, either:*

- \mathcal{E} is not k -expressive for any $k > \sum_{i=0}^m \frac{|X|! |Y|^i}{(|X|-i)!}$, or
- for every CEGIS-based learner \mathcal{L} , there exists a satisfiable SyGuS problem $S = \langle f_{x \rightarrow Y} | \phi, \mathcal{E} \rangle_{\mathbb{T}}$ such that S is not m -learnable by \mathcal{L} . Moreover, there exists a different CEGIS-based learner for which S is m -learnable.

Theorem 2 (NFL in CEGIS-based SyGuS on Countably Infinite Sets). *Let X be an arbitrary countably infinite set, Y be an arbitrary finite or countably infinite set, \mathbb{T} be a theory that supports equality, \mathcal{E} be a grammar over \mathbb{T} , and m be an integer such that $m \geq 0$. Then, either:*

- \mathcal{E} is not k -expressive for any $k > \aleph_0$, where $\aleph_0 \stackrel{\text{def}}{=} |\mathbb{N}|$, or
- for every CEGIS-based learner \mathcal{L} , there exists a satisfiable SyGuS problem $S = \langle f_{x \rightarrow Y} | \phi, \mathcal{E} \rangle_{\mathbb{T}}$ such that S is not m -learnable by \mathcal{L} . Moreover, there exists a different CEGIS-based learner for which S is m -learnable.

3.3 Overfitting

Last, we relate the above theory to the notion of *overfitting* from ML. In the context of SyGuS, overfitting can potentially occur whenever there are multiple candidate expressions that are consistent with a given set of examples. Some of these expressions may not generalize to satisfy the specification, but the learner has no way to distinguish among them (using just the given set of examples) and so can “guess” incorrectly. We formalize this idea through the following measure:

Definition 7 (Potential for Overfitting). *Given a problem $S = \langle f_{x \rightarrow Y} | \phi, \mathcal{E} \rangle_{\mathbb{T}}$ and a set Z of IO examples for ϕ , we define the potential for overfitting Ω as the number of expressions in \mathcal{E} that are consistent with Z but do not satisfy S , i.e.,*

$$\Omega(S, Z) \stackrel{\text{def}}{=} \begin{cases} |\{e \in \mathcal{E} \mid e \not\models S \wedge \forall \langle x, y \rangle \in Z: e(x) = y\}| & \forall z \in Z: z \models_{\mathbb{T}} \phi \\ \perp & \text{(undefined)} \end{cases} \quad \text{otherwise}$$

Intuitively, a zero potential for overfitting means that overfitting is not possible on the given problem with respect to the given set of examples, because there is no spurious candidate. A positive potential for overfitting means that overfitting is possible, and higher values imply more spurious candidates and hence more potential for a learner to choose the “wrong” expression.

The following theorems connect our notion of overfitting to the earlier NFL theorems by showing that overfitting is inevitable with an expressive enough grammar. The proofs of these theorems can be found in the extended version of this paper [27, Appendix A.2].

Theorem 3 (Overfitting in SyGuS on Finite Sets). *Let X and Y be two arbitrary finite sets, m be an integer such that $0 \leq m < |X|$, \mathbb{T} be a theory that supports equality, and \mathcal{E} be a k -expressive grammar over \mathbb{T} for some $k > \frac{|X|^! \cdot |Y|^m}{m! \cdot (|X| - m)!}$. Then, there exists a satisfiable SyGuS problem $S = \langle f_{x \rightarrow Y} \mid \phi, \mathcal{E} \rangle_{\mathbb{T}}$ such that $\Omega(S, Z) > 0$, for every set Z of m IO examples for ϕ .*

Theorem 4 (Overfitting in SyGuS on Countably Infinite Sets). *Let X be an arbitrary countably infinite set, Y be an arbitrary finite or countably infinite set, \mathbb{T} be a theory that supports equality, and \mathcal{E} be a k -expressive grammar over \mathbb{T} for some $k > \aleph_0$. Then, there exists a satisfiable SyGuS problem $S = \langle f_{x \rightarrow Y} \mid \phi, \mathcal{E} \rangle_{\mathbb{T}}$ such that $\Omega(S, Z) > 0$, for every set Z of m IO examples for ϕ .*

Finally, it is straightforward to show that as the expressiveness of the grammar provided in a SyGuS problem increases, so does its potential for overfitting.

Theorem 5 (Overfitting Increases with Expressiveness). *Let X and Y be two arbitrary sets, \mathbb{T} be an arbitrary theory, \mathcal{E}_1 and \mathcal{E}_2 be grammars over \mathbb{T} such that $\mathcal{E}_1 \subseteq \mathcal{E}_2$, ϕ be an arbitrary specification over \mathbb{T} and a function symbol $f: X \rightarrow Y$, and Z be a set of IO examples for ϕ . Then, we have*

$$\Omega(\langle f_{x \rightarrow Y} \mid \phi, \mathcal{E}_1 \rangle_{\mathbb{T}}, Z) \leq \Omega(\langle f_{x \rightarrow Y} \mid \phi, \mathcal{E}_2 \rangle_{\mathbb{T}}, Z)$$

4 Mitigating Overfitting

Ensemble methods [13] in machine learning (ML) are a standard approach to reduce overfitting. These methods aggregate predictions from several learners to make a more accurate prediction. In this section we propose two approaches, inspired by ensemble methods in ML, for mitigating overfitting in SyGuS. Both are based on the key insight from Sect. 3.3 that synthesis over a subgrammar has a smaller potential for overfitting as compared to that over the original grammar.

4.1 Parallel SyGuS on Multiple Grammars

Our first idea is to run multiple parallel instances of a synthesizer on the same SyGuS problem but with grammars of varying expressiveness. This framework, called PLEARN, is outlined in Algorithm 1. It accepts a synthesis tool T , a SyGuS

Algorithm 1. The PLEARN framework for SyGuS tools.

```

1 func PLEARN ( $\mathcal{T}$ : Synthesis Tool,  $\langle f_{X \rightarrow Y} | \phi, \mathcal{E} \rangle_{\mathbb{T}}$ : Problem,  $\mathcal{E}_{1 \dots p}$ : Subgrammars)
2 ► Requires:  $\forall \mathcal{E}_i \in \mathcal{E}_{1 \dots p} : \mathcal{E}_i \subseteq \mathcal{E}$ 
3 parallel for  $i \leftarrow 1, \dots, p$  do
4    $S_i \leftarrow \langle f_{X \rightarrow Y} | \phi, \mathcal{E}_i \rangle_{\mathbb{T}}$ 
5    $e_i \leftarrow \mathcal{T}(S_i)$ 
6   if  $e_i \neq \perp$  then return  $e_i$ 
7 return  $\perp$ 

```

problem $\langle f_{X \rightarrow Y} | \phi, \mathcal{E} \rangle_{\mathbb{T}}$, and subgrammars $\mathcal{E}_{1 \dots p}$ ⁴ such that $\mathcal{E}_i \subseteq \mathcal{E}$. The **parallel for** construct creates a new thread for each iteration. The loop in PLEARN creates p copies of the SyGuS problem, each with a different grammar from $\mathcal{E}_{1 \dots p}$, and dispatches each copy to a new instance of the tool \mathcal{T} . PLEARN returns the first solution found or \perp if none of the synthesizer instances succeed.

Since each grammar in $\mathcal{E}_{1 \dots p}$ is subsumed by the original grammar \mathcal{E} , any expression found by PLEARN is a solution to the original SyGuS problem. Moreover, from Theorem 5 it is immediate that PLEARN indeed reduces overfitting.

Theorem 6 (PLEARN Reduces Overfitting). *Given a SyGuS problem $S = \langle f_{X \rightarrow Y} | \phi, \mathcal{E} \rangle_{\mathbb{T}}$, if PLEARN is instantiated with S and subgrammars $\mathcal{E}_{1 \dots p}$ such that $\forall \mathcal{E}_i \in \mathcal{E}_{1 \dots p} : \mathcal{E}_i \subseteq \mathcal{E}$, then for each $S_i = \langle f_{X \rightarrow Y} | \phi, \mathcal{E}_i \rangle_{\mathbb{T}}$ constructed by PLEARN, we have that $\Omega(S_i, Z) \leq \Omega(S, Z)$ on any set Z of IO examples for ϕ .*

A key advantage of PLEARN is that it is agnostic to the synthesizer’s implementation. Therefore, existing SyGuS learners can immediately benefit from PLEARN, as we demonstrate in Sect. 5.1. However, running p parallel SyGuS instances can be prohibitively expensive, both computationally and memory-wise. The problem is worsened by the fact that many existing SyGuS tools already use multiple threads, e.g., the SKETCHAC [20] tool spawns 9 threads. This motivates our *hybrid enumeration* technique described next, which is a novel synthesis algorithm that interleaves exploration of multiple grammars in a single thread.

4.2 Hybrid Enumeration

Hybrid enumeration extends the *enumerative synthesis* technique, which enumerates expressions within a given grammar in order of size and returns the first candidate that satisfies the given examples [2]. Our goal is to simulate the behavior of PLEARN with an enumerative synthesizer in a single thread. However, a straightforward interleaving of multiple PLEARN threads would be highly inefficient because of redundancies – enumerating the same expression (which is contained in multiple grammars) multiple times. Instead, we propose a technique that (1) enumerates each expression at most once, and (2) reuses previously enumerated expressions to construct larger expressions.

⁴ We use the shorthand $X_{1, \dots, n}$ to denote the sequence $\langle X_1, \dots, X_n \rangle$.

To achieve this, we extend a widely used [2, 15, 31] synthesis strategy, called *component-based synthesis* [21], wherein the grammar of expressions is induced by a set of components, each of which is a typed operator with a fixed arity. For example, the grammars shown in Fig. 1 are induced by integer components (such as `1`, `+`, `mod`, `=`, etc.) and Boolean components (such as `true`, `and`, `or`, etc.). Below, we first formalize the grammar that is implicit in this synthesis style.

Definition 8 (Component-Based Grammar). *Given a set \mathcal{C} of typed components, we define the component-based grammar \mathcal{E} as the set of all expressions formed by well-typed component application over \mathcal{C} , i.e.,*

$$\mathcal{E} = \{ c(e_1, \dots, e_a) \mid (c : \tau_1 \times \dots \times \tau_a \rightarrow \tau) \in \mathcal{C} \wedge e_1 \dots e_a \subseteq \mathcal{E} \wedge e_1 : \tau_1 \wedge \dots \wedge e_a : \tau_a \}$$

where $e : \tau$ denotes that the expression e has type τ .

We denote the set of all components appearing in a component-based grammar \mathcal{E} as $\text{components}(\mathcal{E})$. Henceforth, we assume that $\text{components}(\mathcal{E})$ is known (explicitly provided by the user) for each \mathcal{E} . We also use $\text{values}(\mathcal{E})$ to denote the subset of nullary components (variables and constants) in $\text{components}(\mathcal{E})$, and $\text{operators}(\mathcal{E})$ to denote the remaining components with positive arities.

The closure property of component-based grammars significantly reduces the overhead of tracking which subexpressions can be combined together to form larger expressions. Given a SyGuS problem over a grammar \mathcal{E} , hybrid enumeration requires a sequence $\mathcal{E}_{1\dots p}$ of grammars such that each \mathcal{E}_i is a component-based grammar and that $\mathcal{E}_1 \subset \dots \subset \mathcal{E}_p \subseteq \mathcal{E}$. Next, we explain how the subset relationship between the grammars enables efficient enumeration of expressions.

Given grammars $\mathcal{E}_1 \subset \dots \subset \mathcal{E}_p$, observe that an expression of size k in \mathcal{E}_i may only contain subexpressions of size $\{1, \dots, (k-1)\}$ belonging to $\mathcal{E}_{1\dots i}$. This allows us to enumerate expressions in an order such that each subexpression e is synthesized (and cached) before any expressions that have e as a subexpression. We call an enumeration order that ensures this property a *well order*.

Definition 9 (Well Order). *Given arbitrary grammars $\mathcal{E}_{1\dots p}$, we say that a strict partial order \triangleleft on $\mathcal{E}_{1\dots p} \times \mathbb{N}$ is a well order iff*

$$\forall \mathcal{E}_a, \mathcal{E}_b \in \mathcal{E}_{1\dots p} : \forall k_1, k_2 \in \mathbb{N} : [\mathcal{E}_a \subseteq \mathcal{E}_b \wedge k_1 < k_2] \implies (\mathcal{E}_a, k_1) \triangleleft (\mathcal{E}_b, k_2)$$

Motivated by Theorem 5, our implementation of hybrid enumeration uses a particular well order that incrementally increases the expressiveness of the space of expressions. For a rough measure of the expressiveness (Definition 5) of a pair (\mathcal{E}, k) , i.e., the set of expressions of size k in a given grammar \mathcal{E} , we simply overapproximate the number of syntactically distinct expressions:

Theorem 7. *Let $\mathcal{E}_{1\dots p}$ be component-based grammars and $\mathcal{C}_i = \text{components}(\mathcal{E}_i)$. Then, the following strict partial order \triangleleft_* on $\mathcal{E}_{1\dots p} \times \mathbb{N}$ is a well order*

$$\forall \mathcal{E}_a, \mathcal{E}_b \in \mathcal{E}_{1\dots p} : \forall m, n \in \mathbb{N} : (\mathcal{E}_a, m) \triangleleft_* (\mathcal{E}_b, n) \iff |\mathcal{C}_a|^m < |\mathcal{C}_b|^n$$

We now describe the main hybrid enumeration algorithm, which is listed in Algorithm 2. The HENUM function accepts a SyGuS problem $\langle f_{x \rightarrow Y} | \phi, \mathcal{E} \rangle_{\mathbb{T}}$, a set $\mathcal{E}_{1 \dots p}$ of component-based grammars such that $\mathcal{E}_1 \subset \dots \subset \mathcal{E}_p \subseteq \mathcal{E}$, a well order \triangleleft , and an upper bound $q \geq 0$ on the size of expressions to enumerate. In lines 4–8, we first enumerate all values and cache them as expressions of size one. In general $C[j, k][\tau]$ contains expressions of type τ and size k from $\mathcal{E}_j \setminus \mathcal{E}_{j-1}$. In line 9 we sort (grammar, size) pairs in some total order consistent with \triangleleft . Finally, in lines 10–20, we iterate over each pair (\mathcal{E}_j, k) and each operator from $\mathcal{E}_{1 \dots j}$ and invoke the DIVIDE procedure (Algorithm 3) to carefully choose the operator’s argument subexpressions ensuring (1) *correctness* – their sizes sum up to $k - 1$, (2) *efficiency* – expressions are enumerated at most once, and (3) *completeness* – all expressions of size k in \mathcal{E}_j are enumerated.

The DIVIDE algorithm generates a set of locations for selecting arguments to an operator. Each location is a pair (x, y) indicating that any expression from $C[x, y][\tau]$ can be an argument, where τ is the argument type required by the operator. DIVIDE accepts an arity a for an operator o , a size budget q , the index l of the least-expressive grammar containing o , the index j of the least-expressive grammar that should contain the constructed expressions of the form $o(e_1, \dots, e_a)$, and an accumulator α that stores the list of argument locations. In lines 7–9, the size budget is recursively divided among $a - 1$ locations. In each recursive step, the upper bound $(q - a + 1)$ on v ensures that we have a size budget of at least $q - (q - a + 1) = a - 1$ for the remaining $a - 1$ locations. This results in a call tree such that the accumulator α at each leaf node contains the locations from which to select the last $a - 1$ arguments, and we are left with some size budget $q \geq 1$ for the first argument e_1 . Finally in lines 4–5, we carefully select the locations for e_1 to ensure that $o(e_1, \dots, e_a)$ has not been synthesized before—either $o \in \text{components}(\mathcal{E}_j)$ or at least one argument belongs to $\mathcal{E}_j \setminus \mathcal{E}_{j-1}$.⁵

We conclude by stating some desirable properties satisfied by HENUM. Their proofs are provided in the extended version of this paper [27, Appendix A.3].

Theorem 8 (HENUM is Complete up to Size q). *Given a SyGuS problem $S = \langle f_{x \rightarrow Y} | \phi, \mathcal{E} \rangle_{\mathbb{T}}$, let $\mathcal{E}_{1 \dots p}$ be component-based grammars over theory \mathbb{T} such that $\mathcal{E}_1 \subset \dots \subset \mathcal{E}_p = \mathcal{E}$, \triangleleft be a well order on $\mathcal{E}_{1 \dots p} \times \mathbb{N}$, and $q \geq 0$ be an upper bound on size of expressions. Then, $\text{HENUM}(S, \mathcal{E}_{1 \dots p}, \triangleleft, q)$ will eventually find a satisfying expression if there exists one with size $\leq q$.*

Theorem 9 (HENUM is Efficient). *Given a SyGuS problem $S = \langle f_{x \rightarrow Y} | \phi, \mathcal{E} \rangle_{\mathbb{T}}$, let $\mathcal{E}_{1 \dots p}$ be component-based grammars over theory \mathbb{T} such that $\mathcal{E}_1 \subset \dots \subset \mathcal{E}_p \subseteq \mathcal{E}$, \triangleleft be a well order on $\mathcal{E}_{1 \dots p} \times \mathbb{N}$, and $q \geq 0$ be an upper bound on size of expressions. Then, $\text{HENUM}(S, \mathcal{E}_{1 \dots p}, \triangleleft, q)$ will enumerate each distinct expression at most once.*

⁵ We use \diamond as the `cons` operator for sequences, e.g., $x \diamond \langle y, z \rangle = \langle x, y, z \rangle$.

Algorithm 2. Hybrid enumeration to combat overfitting in SyGuS

```

1  func HENUM ( $\langle f_{X \rightarrow Y} | \phi, \mathcal{E} \rangle_{\mathbb{T}}$  : Problem,  $\mathcal{E}_{1\dots p}$ : Grammars,  $\triangleleft$ : WO,  $q$ : Max. Size)
2  ► Requires: component-based grammars  $\mathcal{E}_1 \subset \dots \subset \mathcal{E}_p \subseteq \mathcal{E}$  and  $v$  as the input variable
3   $C \leftarrow \{\}$ 
4  for  $i \leftarrow 1$  to  $p$  do
5     $V \leftarrow$  if  $i = 1$  then  $\text{values}(\mathcal{E}_1)$  else [ $\text{values}(\mathcal{E}_i) \setminus \text{values}(\mathcal{E}_{i-1})$ ]
6    for each  $(e : \tau) \in V$  do
7       $C[i, 1][\tau] \leftarrow C[i, 1][\tau] \cup \{e\}$ 
8      if  $\forall x \in X : \phi(\lambda v. e, x)$  then return  $\lambda v. e$ 
9     $R \leftarrow \text{SORT}(\triangleleft, \mathcal{E}_{1\dots p} \times \{2, \dots, q\})$ 
10   for  $i \leftarrow 1$  to  $|R|$  do
11      $(\mathcal{E}_j, k) \leftarrow R[i]$ 
12     for  $l \leftarrow 1$  to  $j$  do
13        $O \leftarrow$  if  $l = 1$  then  $\text{operators}(\mathcal{E}_1)$  else [ $\text{operators}(\mathcal{E}_l) \setminus \text{operators}(\mathcal{E}_{l-1})$ ]
14       for each  $(o : \tau_1 \times \dots \times \tau_a \rightarrow \tau) \in O$  do
15          $L \leftarrow \text{DIVIDE}(a, k - 1, l, j, \langle \rangle)$ 
16         for each  $\langle (x_1, y_1), \dots, (x_a, y_a) \rangle \in L$  do
17           for each  $e_{1\dots a} \in C[x_1, y_1][\tau_1] \times \dots \times C[x_a, y_a][\tau_a]$  do
18              $e \leftarrow o(e_1, \dots, e_a)$ 
19              $C[j, k][\tau] \leftarrow C[j, k][\tau] \cup \{e\}$ 
20             if  $\forall x \in X : \phi(\lambda v. e, x)$  then return  $\lambda v. e$ 
21   return  $\perp$ 

```

Algorithm 3. An algorithm to divide a given size budget among subexpressions⁵

```

1  func DIVIDE ( $a$ : Arity,  $q$ : Size,  $l$ : Op. Level,  $j$ : Expr. Level,  $\alpha$ : Accumulated Args.)
2  ► Requires:  $1 \leq a \leq q \wedge l \leq j$ 
3  if  $a = 1$  then
4    if  $l = j \vee \exists \langle x, y \rangle \in \alpha : x = j$  then return  $\{(1, q) \diamond \alpha, \dots, (j, q) \diamond \alpha\}$ 
5    return  $\{(j, q) \diamond \alpha\}$ 
6   $L = \{\}$ 
7  for  $u \leftarrow 1$  to  $j$  do
8    for  $v \leftarrow 1$  to  $(q - a + 1)$  do
9       $L \leftarrow L \cup \text{DIVIDE}(a - 1, q - v, l, j, (u, v) \diamond \alpha)$ 
10   return  $L$ 

```

5 Experimental Evaluation

In this section we empirically evaluate PLEARN and HENUM. Our evaluation uses a set of 180 synthesis benchmarks,⁶ consisting of all 127 official benchmarks from the Inv track of 2018 SyGuS competition [4] augmented with benchmarks from the 2018 Software Verification competition (SV-Comp) [8] and challenging verification problems proposed in prior work [9, 10]. All these synthesis tasks are

⁶ All benchmarks are available at <https://github.com/SaswatPadhi/LoopInvGen>.

defined over integer and Boolean values, and we evaluate them with the six grammars described in Fig. 1. We have omitted benchmarks from other tracks of the SyGuS competition as they either require us to construct $\mathcal{E}_{1\dots p}$ (Sect. 4) by hand or lack verification oracles. All our experiments use an 8-core Intel® Xeon® E5 machine clocked at 2.30 GHz with 32 GB memory running Ubuntu® 18.04.

5.1 Robustness of PLEARN

For five state-of-the-art SyGuS solvers – (a) LOOPINVENT [29], (b) CVC4 [7, 33], (c) STOCH [3, III F], (d) SKETCHAC [8, 20], and (e) EU SOLVER [5] – we have compared the performance across various grammars, with and without the PLEARN framework (Algorithm 1). In this framework, to solve a SyGuS problem with the p^{th} expressiveness level from our six integer-arithmetic grammars (see Fig. 1), we run p independent parallel instances of a SyGuS tool, each with one of the first p grammars. For example, to solve a SyGuS problem with the Polyhedra grammar, we run four instances of a solver with the Equalities, Intervals, Octagons and Polyhedra grammars. We evaluate these runs for each tool, for each of the 180 benchmarks and for each of the six expressiveness levels.

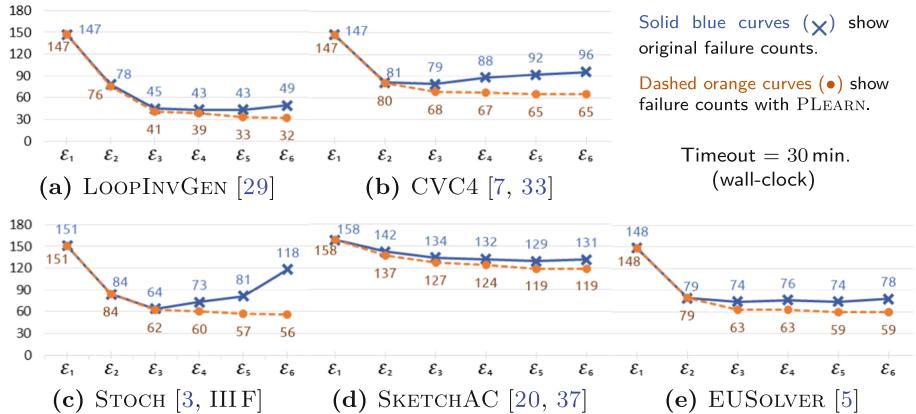


Fig. 6. The number of failures on increasing grammar expressiveness, for state-of-the-art SyGuS tools, with and without the PLEARN framework (Algorithm 1)

Figure 6 summarizes our findings. Without PLEARN the number of failures initially decreases and then increases across all solvers, as grammar expressiveness increases. However, with PLEARN the tools incur fewer failures at a given level of expressiveness, and there is a trend of *decreased* failures with increased expressiveness. Thus, we have demonstrated that PLEARN is an effective measure to mitigate overfitting in SyGuS tools and significantly improve their performance.

5.2 Performance of Hybrid Enumeration

To evaluate the performance of hybrid enumeration, we augment an existing synthesis engine with HENUM (Algorithm 2). We modify our LOOPINVGEN tool [29], which is the best-performing SyGuS synthesizer from Fig. 6. Internally, LOOPINVGEN leverages ESCHER [2], an enumerative synthesizer, which we replace with HENUM. We make no other changes to LOOPINVGEN. We evaluate the performance and resource usage of this solver, LOOPINVGEN+HE, relative to the original LOOPINVGEN with and without PLEARN (Algorithm 1).

Performance. In Fig. 7(a), we show the number of failures across our six grammars for LOOPINVGEN, LOOPINVGEN+HE and LOOPINVGEN with PLEARN, over our 180 benchmarks. LOOPINVGEN+HE has a significantly lower failure rate than LOOPINVGEN, and the number of failures decreases with grammar expressiveness. Thus, hybrid enumeration is a good proxy for PLEARN.

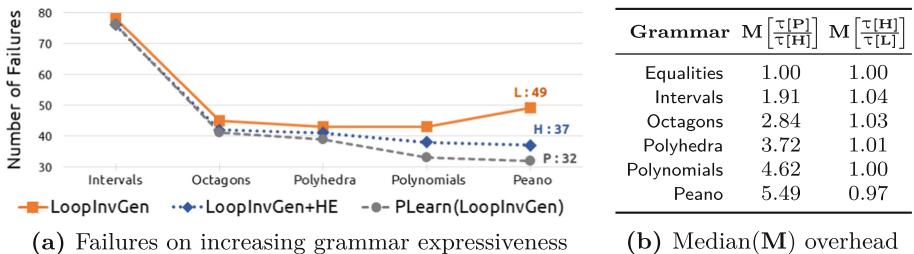


Fig. 7. **L**=LOOPINVGEN, **H**=LOOPINVGEN+HE, **P**=PLEARN (LOOPINVGEN). **H** is not only significantly robust against increasing grammar expressiveness, but it also has a smaller total-time cost (τ) than **P** and a negligible overhead over **L**.

Resource Usage. To estimate how computationally expensive each solver is, we compare their *total-time cost* (τ). Since LOOPINVGEN and LOOPINVGEN+HE are single-threaded, for them we simply use the wall-clock time for synthesis as the total-time cost. However, for PLEARN with p parallel instances of LOOPINVGEN, we consider the total-time cost as p times the wall-clock time for synthesis.

In Fig. 7(b), we show the median overhead (ratio of τ) incurred by PLEARN over LOOPINVGEN+HE and LOOPINVGEN+HE over LOOPINVGEN, at various expressiveness levels. As we move to grammars of increasing expressiveness, the total-time cost of PLEARN increases significantly, while the total-time cost of LOOPINVGEN+HE essentially matches that of LOOPINVGEN.

5.3 Competition Performance

Finally, we evaluate the performance of LOOPINVGEN+HE on the benchmarks from the Inv track of the 2018 SyGuS competition [4], against the official winning solver, which we denote LIG [28]—a version of LOOPINVGEN [29] that has been extensively tuned for this track. In the competition, there are some invariant-synthesis problems where the postcondition itself is a satisfying expression.

LIG starts with the postcondition as the first candidate and is extremely fast on such programs. For a fair comparison, we added this heuristic to LOOPINVGEN+HE as well. No other change was made to LOOPINVGEN+HE.

LOOPINVGEN solves 115 benchmarks in a total of 2191 seconds whereas LOOPINVGEN+HE solves 117 benchmarks in 429 seconds, for a mean speedup of over 5 \times . Moreover, no entrants to the competition could solve [4] the two additional benchmarks (`gcnr_tacas08` and `fib_20`) that LOOPINVGEN+HE solves.

6 Related Work

The most closely related work to ours investigates overfitting for verification tools [36]. Our work differs from theirs in several respects. First, we address the problem of overfitting in CEGIS-based synthesis. Second, we formally define overfitting and prove that all synthesizers must suffer from it, whereas they only observe overfitting empirically. Third, while they use cross-validation to combat overfitting in tuning a specific hyperparameter of a verifier, our approach is to search for solutions at different expressiveness levels.

The general problem of efficiently searching a large space of programs for synthesis has been explored in prior work. Lee et al. [24] use a probabilistic model, learned from known solutions to synthesis problems, to enumerate programs in order of their likelihood. Other approaches employ type-based pruning of large search spaces [26, 32]. These techniques are orthogonal to, and may be combined with, our approach of exploring grammar subsets.

Our results are widely applicable to existing SyGuS tools, but some tools fall outside our purview. For instance, in programming-by-example (PBE) systems [18, §7], the specification consists of a set of input-output examples. Since any program that meets the given examples is a valid satisfying expression, our notion of overfitting does not apply to such tools. However in a recent work, Inala and Singh [19] show that incrementally increasing expressiveness can also aid PBE systems. They report that searching within increasingly expressive grammar subsets requires significantly fewer examples to find expressions that generalize better over unseen data. Other instances where the synthesizers can have a free lunch, i.e., always generate a solution with a small number of counterexamples, include systems that use grammars with limited expressiveness [16, 21, 35].

Our paper falls in the category of formal results about SyGuS. In one such result, Jha and Seshia [22] analyze the effects of different kinds of counterexamples and of providing bounded versus unbounded memory to learners. Notably, they do not consider variations in “concept classes” or “program templates,” which are precisely the focus of our study. Therefore, our results are complementary: we treat counterexamples and learners as opaque and instead focus on grammars.

7 Conclusion

Program synthesis is a vibrant research area; new and better synthesizers are being built each year. This paper investigates a general issue that affects all

CEGIS-based SyGuS tools. We recognize the problem of overfitting, formalize it, and identify the conditions under which it must occur. Furthermore, we provide mitigating measures for overfitting that significantly improve the existing tools.

Acknowledgement. We thank Guy Van den Broeck and the anonymous reviewers for helpful feedback for improving this work, and the organizers of the SyGuS competition for making the tools and benchmarks publicly available.

This work was supported in part by the National Science Foundation (NSF) under grants CCF-1527923 and CCF-1837129. The lead author was also supported by an internship and a PhD Fellowship from Microsoft Research.

References

1. The SyGuS Competition (2019). <http://sygus.org/comp/>. Accessed 10 May 2019
2. Albarghouthi, A., Gulwani, S., Kincaid, Z.: Recursive program synthesis. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 934–950. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_67
3. Alur, R., et al.: Syntax-guided synthesis. In: Formal Methods in Computer-Aided Design, FMCAD, pp. 1–8. IEEE (2013). <http://ieeexplore.ieee.org/document/6679385/>
4. Alur, R., Fisman, D., Padhi, S., Singh, R., Udupa, A.: SyGuS-Comp 2018: Results and Analysis. CoRR abs/1904.07146 (2019). <http://arxiv.org/abs/1904.07146>
5. Alur, R., Radhakrishna, A., Udupa, A.: Scaling enumerative program synthesis via divide and conquer. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10205, pp. 319–336. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_18
6. Alur, R., Singh, R., Fisman, D., Solar-Lezama, A.: Search-based program synthesis. Commun. ACM **61**(12), 84–93 (2018). <https://doi.org/10.1145/3208071>
7. Barrett, C., et al.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_14
8. Beyer, D.: Software verification with validation of results. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10206, pp. 331–349. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54580-5_20
9. Bounov, D., DeRossi, A., Menarini, M., Griswold, W.G., Lerner, S.: Inferring loop invariants through gamification. In: Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems, CHI, p. 231. ACM (2018). <https://doi.org/10.1145/3173574.3173805>
10. Bradley, A.R., Manna, Z., Sipma, H.B.: The polyranking principle. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 1349–1361. Springer, Heidelberg (2005). https://doi.org/10.1007/11523468_109
11. Cousot, P., Cousot, R.: Static determination of dynamic properties of generalized type unions. In: Language Design for Reliable Software, pp. 77–94 (1977). <https://doi.org/10.1145/800022.808314>
12. Cousot, P., Halbwachs, N.: Automatic Discovery of Linear Restraints Among Variables of a Program. In: Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages. pp. 84–96. ACM Press (1978), <https://doi.org/10.1145/512760.512770>

13. Dietterich, T.G.: Ensemble methods in machine learning. In: Kittler, J., Roli, F. (eds.) MCS 2000. LNCS, vol. 1857, pp. 1–15. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-45014-9_1
14. Ezudheen, P., Neider, D., D’Souza, D., Garg, P., Madhusudan, P.: Horn-ICE learning for synthesizing invariants and contracts. PACMPL **2**(OOPSLA), 131:1–131:25 (2018). <https://doi.org/10.1145/3276501>
15. Feng, Y., Martins, R., Geffen, J.V., Dillig, I., Chaudhuri, S.: Component-based synthesis of table consolidation and transformation tasks from examples. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI, pp. 422–436. ACM (2017). <https://doi.org/10.1145/3062341.3062351>
16. Godefroid, P., Taly, A.: Automated synthesis of symbolic instruction encodings from I/O samples. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI, pp. 441–452. ACM (2012). <https://doi.org/10.1145/2254064.2254116>
17. Gulwani, S., Jovic, N.: Program verification as probabilistic inference. In: Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL, pp. 277–289. ACM (2007). <https://doi.org/10.1145/1190216.1190258>
18. Gulwani, S., Polozov, O., Singh, R.: Program synthesis. Found. Trends Program. Lang. **4**(1–2), 1–119 (2017). <https://doi.org/10.1561/2500000010>
19. Inala, J.P., Singh, R.: WebRelate: Integrating Web Data with Spreadsheets using Examples. PACMPL **2**(POPL), 2:1–2:28 (2018). <https://doi.org/10.1145/3158090>
20. Jeon, J., Qiu, X., Solar-Lezama, A., Foster, J.S.: Adaptive concretization for parallel program synthesis. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9207, pp. 377–394. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21668-3_22
21. Jha, S., Gulwani, S., Seshia, S.A., Tiwari, A.: Oracle-guided component-based program synthesis. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering. ICSE, vol. 1, pp. 215–224. ACM (2010). <https://doi.org/10.1145/1806799.1806833>
22. Jha, S., Seshia, S.A.: A theory of formal synthesis via inductive learning. Acta Informatica **54**(7), 693–726 (2017). <https://doi.org/10.1007/s00236-017-0294-5>
23. Le, X.D., Chu, D., Lo, D., Le Goues, C., Visser, W.: S3: syntax- and semantic-guided repair synthesis via programming by examples. In: Proceedings of the 11th Joint Meeting on Foundations of Software Engineering. ESEC/FSE, pp. 593–604. ACM (2017). <https://doi.org/10.1145/3106237.3106309>
24. Lee, W., Heo, K., Alur, R., Naik, M.: Accelerating search-based program synthesis using learned probabilistic models. In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, pp. 436–449. ACM (2018). <https://doi.org/10.1145/3192366.3192410>
25. Miné, A.: The octagon abstract domain. In: Proceedings of the Eighth Working Conference on Reverse Engineering, WCRE, p. 310. IEEE Computer Society (2001). <https://doi.org/10.1109/WCRE.2001.957836>
26. Osera, P., Zdancewic, S.: Type-and-example-directed program synthesis. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI, pp. 619–630. ACM (2015). <https://doi.org/10.1145/2737924.2738007>
27. Padhi, S., Millstein, T., Nori, A., Sharma, R.: Overfitting in Synthesis: Theory and Practice. CoRR abs/1905.07457 (2019). <https://arxiv.org/pdf/1905.07457>

28. Padhi, S., Sharma, R., Millstein, T.: LoopInvGen: A Loop Invariant Generator based on Precondition Inference. CoRR abs/1707.02029 (2018). <http://arxiv.org/abs/1707.02029>
29. Padhi, S., Sharma, R., Millstein, T.D.: Data-driven precondition inference with learned features. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI, pp. 42–56. ACM (2016). <https://doi.org/10.1145/2908080.2908099>
30. Peano, G.: Calcolo geometrico secondo l’Ausdehnungslehre di H. Grassmann: preceduto dalla operazioni della logica deduttiva, vol. 3. Fratelli Bocca (1888)
31. Perelman, D., Gulwani, S., Grossman, D., Provost, P.: Test-driven synthesis. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI, pp. 408–418. ACM (2014). <https://doi.org/10.1145/2594291.2594297>
32. Polikarpova, N., Kuraj, I., Solar-Lezama, A.: Program synthesis from polymorphic refinement types. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI, pp. 522–538. ACM (2016). <https://doi.org/10.1145/2908080.2908093>
33. Reynolds, A., Deters, M., Kuncak, V., Tinelli, C., Barrett, C.: Counterexample-guided quantifier instantiation for synthesis in SMT. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9207, pp. 198–216. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21668-3_12
34. Shalev-Shwartz, S., Ben-David, S.: Understanding Machine Learning: From Theory to Algorithms. Cambridge University Press, Cambridge (2014)
35. Sharma, R., Gupta, S., Hariharan, B., Aiken, A., Liang, P., Nori, A.V.: A data driven approach for algebraic loop invariants. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 574–592. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_31
36. Sharma, R., Nori, A.V., Aiken, A.: Bias-variance tradeoffs in program analysis. In: The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL, pp. 127–138. ACM (2014). <https://doi.org/10.1145/2535838.2535853>
37. Solar-Lezama, A.: Program sketching. STTT **15**(5–6), 475–495 (2013)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter’s Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter’s Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Proving Unrealizability for Syntax-Guided Synthesis

Qinheping Hu^{1(✉)}, Jason Breck¹, John Cyphert¹, Loris D’Antoni¹,
and Thomas Reps^{1,2}

¹ University of Wisconsin-Madison, Madison, USA
qhu28@wisc.edu

² GrammaTech, Inc., Ithaca, USA

Abstract. We consider the problem of automatically establishing that a given syntax-guided-synthesis (SyGuS) problem is *unrealizable* (i.e., has no solution). Existing techniques have quite limited ability to establish unrealizability for general SYGUS instances in which the grammar describing the search space contains infinitely many programs. By encoding the synthesis problem’s grammar G as a nondeterministic program P_G , we reduce the unrealizability problem to a reachability problem such that, if a standard program-analysis tool can establish that a certain assertion in P_G always holds, then the synthesis problem is unrealizable.

Our method can be used to augment existing SyGuS tools so that they can establish that a successfully synthesized program q is *optimal* with respect to some syntactic cost—e.g., q has the fewest possible if-then-else operators. Using known techniques, grammar G can be transformed to generate the set of all programs with lower costs than q —e.g., fewer conditional expressions. Our algorithm can then be applied to show that the resulting synthesis problem is unrealizable. We implemented the proposed technique in a tool called NOPE. NOPE can prove unrealizability for 59/132 variants of existing linear-integer-arithmetic SYGUS benchmarks, whereas all existing SYGUS solvers lack the ability to prove that these benchmarks are unrealizable, and time out on them.

1 Introduction

The goal of program synthesis is to find a program in some search space that meets a specification—e.g., satisfies a set of examples or a logical formula. Recently, a large family of synthesis problems has been unified into a framework called *syntax-guided synthesis* (SyGuS). A SyGuS problem is specified by a regular-tree grammar that describes the search space of programs, and a logical formula that constitutes the behavioral specification. Many synthesizers now support a specific format for SyGuS problems [1], and compete in annual synthesis competitions [2]. Thanks to these competitions, these solvers are now quite mature and are finding a wealth of applications [9].

Consider the SyGuS problem to synthesize a function f that computes the maximum of two variables x and y , denoted by $(\psi_{\max 2}(f, x, y), G_1)$. The goal is to

© The Author(s) 2019

I. Dillig and S. Tasiran (Eds.): CAV 2019, LNCS 11561, pp. 335–352, 2019.

https://doi.org/10.1007/978-3-030-25540-4_18

create e_f —an expression-tree for f —where e_f is in the language of the following regular-tree grammar G_1 :

$$\begin{aligned} \text{Start} &::= \text{Plus}(\text{Start}, \text{Start}) \mid \text{IfThenElse}(\text{BExpr}, \text{Start}, \text{Start}) \mid x \mid y \mid 0 \mid 1 \\ \text{BExpr} &::= \text{GreaterThan}(\text{Start}, \text{Start}) \mid \text{Not}(\text{BExpr}) \mid \text{And}(\text{BExpr}, \text{BExpr}) \end{aligned}$$

and $\forall x, y. \psi_{\max 2}(\llbracket e_f \rrbracket, x, y)$ is valid, where $\llbracket e_f \rrbracket$ denotes the meaning of e_f , and

$$\psi_{\max 2}(f, x, y) := f(x, y) \geq x \wedge f(x, y) \geq y \wedge (f(x, y) = x \vee f(x, y) = y).$$

SYGUS solvers can easily find a solution, such as

$$e := \text{IfThenElse}(\text{GreaterThan}(x, y), x, y).$$

Although many solvers can now find solutions efficiently to many SYGUS problems, there has been effectively no work on the much harder task of proving that a given SYGUS problem is *unrealizable*—i.e., it does not admit a solution. For example, consider the SYGUS problem $(\psi_{\max 2}(f, x, y), G_2)$, where G_2 is the more restricted grammar with if-then-else operators and conditions stripped out:

$$\text{Start} ::= \text{Plus}(\text{Start}, \text{Start}) \mid x \mid y \mid 0 \mid 1$$

This SYGUS problem does *not* have a solution, because no expression generated by G_2 meets the specification.¹ However, to the best of our knowledge, current SYGUS solvers cannot prove that such a SYGUS problem is unrealizable.²

A key property of the previous example is that the grammar is infinite. When such a SYGUS problem is realizable, any search technique that systematically explores the infinite search space of possible programs will eventually identify a solution to the synthesis problem. In contrast, proving that a problem is unrealizable requires showing that *every* program in the *infinite* search space *fails to satisfy* the specification. This problem is in general undecidable [6]. Although we cannot hope to have an algorithm for establishing unrealizability, the challenge is to find a technique that succeeds for the kinds of problems encountered in practice. Existing synthesizers can detect the absence of a solution in certain cases (e.g., because the grammar is finite, or is infinite but only generate a finite number of functionally distinct programs). However, in practice, as our

¹ Grammar G_2 only generates terms that are equivalent to some linear function of x and y ; however, the maximum function cannot be described by a linear function.

² The synthesis problem presented above is one that is generated by a recent tool called QSYGUS, which extends SYGUS with quantitative syntactic objectives [10]. The advantage of using quantitative objectives in synthesis is that they can be used to produce higher-quality solutions—e.g., smaller, more readable, more efficient, etc. The synthesis problem $(\psi_{\max 2}(f, x, y), G_2)$ arises from a QSYGUS problem in which the goal is to produce an expression that (i) satisfies the specification $\psi_{\max 2}(f, x, y)$, and (ii) uses the smallest possible number of if-then-else operators. Existing SYGUS solvers can easily produce a solution that uses one if-then-else operator, but cannot prove that no better solution exists—i.e., $(\psi_{\max 2}(f, x, y), G_2)$ is unrealizable.

experiments show, this ability is limited—no existing solver was able to show unrealizability for any of the examples considered in this paper.

In this paper, we present a technique for proving that a possibly infinite SYGUS problem is unrealizable. Our technique builds on two ideas.

1. We observe that unrealizability can often be proven using *finitely many input examples*. In Sect. 2, we show how the example discussed above can be proven to be unrealizable using four input examples— $(0, 0)$, $(0, 1)$, $(1, 0)$, and $(1, 1)$.
2. We devise a way to encode a SYGUS problem $(\psi(f, \bar{x}), G)$ over a finite set of examples E as a *reachability problem in a recursive program* $P[G, E]$. In particular, the program that we construct has an assertion that holds if and only if the given SYGUS problem is unrealizable. Consequently, *unrealizability* can be proven by establishing that the assertion always holds. This property can often be established by a conventional program-analysis tool.

The encoding mentioned in item 2 is non-trivial for three reasons. The following list explains each issue, and sketches how they are addressed

(1) *Infinitely many terms*. We need to model the infinitely many terms generated by the grammar of a given synthesis problem $(\psi(f, \bar{x}), G)$.

To address this issue, we use non-determinism and recursion, and give an encoding $P[G, E]$ such that (i) each non-deterministic path p in the program $P[G, E]$ corresponds to a possible expression e_p that G can generate, and (ii) for each expression e that G can generate, there is a path p_e in $P[G, E]$. (There is an isomorphism between paths and the expression-trees of G)

(2) *Nondeterminism*. We need the computation performed along each path p in $P[G, E]$ to mimic the execution of expression e_p . Because the program uses non-determinism, we need to make sure that, for a given path p in the program $P[G, E]$, computational steps are carried out that mimic the evaluation of e_p for each of the finitely many example inputs in E .

We address this issue by threading the expression-evaluation computations associated with each example in E through the *same* non-deterministic choices.

(3) *Complex Specifications*. We need to handle specifications that allow for nested calls of the programs being synthesized.

For instance, consider the specification $f(f(x)) = x$. To handle this specification, we introduce a new variable y and rewrite the specification as $f(x) = y \wedge f(y) = x$. Because y is now also used as an input to f , we will thread both the computations of x and y through the non-deterministic recursive program.

Our work makes the following contributions:

- We reduce the SYGUS unrealizability problem to a reachability problem to which standard program-analysis tools can be applied (Sects. 2 and 4).
- We observe that, for many SYGUS problems, unrealizability can be proven using *finitely many input examples*, and use this idea to apply the Counter-Example-Guided Inductive Synthesis (CEGIS) algorithm to the problem of proving unrealizability (Sect. 3).

- We give an encoding of a SYGUS problem $(\psi(f, \bar{x}), G)$ over a finite set of examples E as a reachability problem in a nondeterministic recursive program $P[G, E]$, which has the following property: if a certain assertion in $P[G, E]$ always holds, then the synthesis problem is unrealizable (Sect. 4).
- We implement our technique in a tool NOPE using the ESolver synthesizer [2] as the SYGUS solver and the SeaHorn tool [8] for checking reachability. NOPE is able to establish unrealizability for 59 out of 132 variants of benchmarks taken from the SYGUS competition. In particular, NOPE solves all benchmarks with no more than 15 productions in the grammar and requiring no more than 9 input examples for proving unrealizability. Existing SYGUS solvers lack the ability to prove that these benchmarks are unrealizable, and time out on them.

Section 6 discusses related work. Some additional technical material, proofs, and full experimental results are given in [13].

2 Illustrative Example

In this section, we illustrate the main components of our framework for establishing the unrealizability of a SYGUS problem.

Consider the SYGUS problem to synthesize a function f that computes the maximum of two variables x and y , denoted by $(\psi_{\max 2}(f, x, y), G_1)$. The goal is to create e_f —an expression-tree for f —where e_f is in the language of the following regular-tree grammar G_1 :

$$\begin{aligned} \text{Start} &::= \text{Plus}(\text{Start}, \text{Start}) \mid \text{IfThenElse}(\text{BExpr}, \text{Start}, \text{Start}) \mid x \mid y \mid 0 \mid 1 \\ \text{BExpr} &::= \text{GreaterThan}(\text{Start}, \text{Start}) \mid \text{Not}(\text{BExpr}) \mid \text{And}(\text{BExpr}, \text{BExpr}) \end{aligned}$$

and $\forall x, y. \psi_{\max 2}(\llbracket e_f \rrbracket, x, y)$ is valid, where $\llbracket e_f \rrbracket$ denotes the meaning of e_f , and

$$\psi_{\max 2}(f, x, y) := f(x, y) \geq x \wedge f(x, y) \geq y \wedge (f(x, y) = x \vee f(x, y) = y).$$

SYGUS solvers can easily find a solution, such as

$$e := \text{IfThenElse}(\text{GreaterThan}(x, y), x, y).$$

Although many solvers can now find solutions efficiently to many SYGUS problems, there has been effectively no work on the much harder task of proving that a given SYGUS problem is *unrealizable*—i.e., it does not admit a solution. For example, consider the SYGUS problem $(\psi_{\max 2}(f, x, y), G_2)$, where G_2 is the more restricted grammar with if-then-else operators and conditions stripped out:

$$\text{Start} ::= \text{Plus}(\text{Start}, \text{Start}) \mid x \mid y \mid 0 \mid 1$$

This SYGUS problem does *not* have a solution, because no expression generated by G_2 meets the specification.³ However, to the best of our knowledge, current

³ Grammar G_2 generates all linear functions of x and y , and hence generates an infinite number of functionally distinct programs; however, the maximum function cannot be described by a linear function.

SYGUS solvers cannot prove that such a SYGUS problem is unrealizable. As an example, we use the problem $(\psi_{\max2}(f, x, y), G_2)$ discussed in Sect. 1, and show how unrealizability can be proven using four input examples: $(0, 0)$, $(0, 1)$, $(1, 0)$, and $(1, 1)$.

```

1 int I_0;
2 void Start(int x_0,int y_0){
3     if(nd()){ // Encodes "Start ::= Plus(Start, Start)"
4         Start(x_0, y_0);
5         int tempL_0 = I_0;
6         Start(x_0, y_0);
7         int tempR_0 = I_0;
8         I_0 = tempL_0 + tempR_0;
9     }
10    else if(nd()) I_0 = x_0; // Encodes "Start ::= x"
11    else if(nd()) I_0 = y_0; // Encodes "Start ::= y"
12    else if(nd()) I_0 = 1; // Encodes "Start ::= 1"
13    else I_0 = 0; // Encodes "Start ::= 0"
14 }
15
16 bool spec(int x, int y, int f){
17     return (f>=x && f>=y && (f==x || f==y))
18 }
19
20 void main(){
21     int x_0 = 0; int y_0 = 1; // Input example (0,1)
22     Start(x_0,y_0);
23     assert(!spec(x_0,y_0,I_0));
24 }
```

Fig. 1. Program $P[G_2, E_1]$ created during the course of proving the unrealizability of $(\psi_{\max2}(f, x, y), G_2)$ using the set of input examples $E_1 = \{(0, 1)\}$.

Our method can be seen as a variant of Counter-Example-Guided Inductive Synthesis (CEGIS), in which the goal is to create a program P in which a certain assertion always holds. Until such a program is created, each round of the algorithm returns a counter-example, from which we extract an additional input example for the original SYGUS problem. On the i^{th} round, the current set of input examples E_i is used, together with the grammar—in this case G_2 —and the specification of the desired behavior— $\psi_{\max2}(f, x, y)$, to create a candidate program $P[G_2, E_i]$. The program $P[G_2, E_i]$ contains an assertion, and a standard program analyzer is used to check whether the assertion always holds.

Suppose that for the SYGUS problem $(\psi_{\max2}(f, x, y), G_2)$ we start with just the one example input $(0, 1)$ —i.e., $E_1 = \{(0, 1)\}$. Figure 1 shows the initial program $P[G_2, E_1]$ that our method creates. The function `spec` implements the predicate $\psi_{\max2}(f, x, y)$. (All of the programs $\{P[G_2, E_i]\}$ use the same function `spec`). The initialization statements “`int x_0 = 0; int y_0 = 1;`” at line (21) in procedure `main` correspond to the input example $(0, 1)$. The recursive procedure `Start` encodes the productions of grammar G_2 . `Start` is non-deterministic; it contains four calls to an external function `nd()`, which returns

a non-deterministically chosen Boolean value. The calls to `nd()` can be understood as controlling whether or not a production is selected from G_2 during a top-down, left-to-right generation of an expression-tree: lines (3)–(8) correspond to “`Start ::= Plus(Start, Start)`,” and lines (10), (11), (12), and (13) correspond to “`Start ::= x`,” “`Start ::= y`,” “`Start ::= 1`,” and “`Start ::= 0`,” respectively. The code in the five cases in the body of `Start` encodes the semantics of the respective production of G_2 ; in particular, the statements that are executed along any execution path of $P[G_2, E_1]$ implement the *bottom-up evaluation of some expression-tree that can be generated by G_2* . For instance, consider the path that visits statements in the following order (for brevity, some statement numbers have been elided):

```
21 22 (Start 3 4 (Start 10 )Start 6 (Start 12 )Start 8 )Start 23, (1)
```

where `(Start` and `)Start` indicate entry to, and return from, procedure `Start`, respectively. Path (1) corresponds to the top-down, left-to-right generation of the expression-tree `Plus(x, 1)`, interleaved with the tree’s bottom-up evaluation.

Note that with path (1), when control returns to `main`, variable `I_0` has the value 1, and thus the assertion at line (23) fails.

A sound program analyzer will discover that some such path exists in the program, and will return the sequence of non-deterministic choices required to follow one such path. Suppose that the analyzer chooses to report path (1); the sequence of choices would be t, f, t, f, f, f, f, t , which can be decoded to create the expression-tree `Plus(x, 1)`. At this point, we have a candidate definition for f : $f = x + 1$. This formula can be checked using an SMT solver to see whether it satisfies the behavioral specification $\psi_{\max 2}(f, x, y)$. In this case, the SMT solver returns “false.” One counter-example that it could return is $(0, 0)$.

At this point, program $P[G_2, E_2]$ would be constructed using both of the example inputs $(0, 1)$ and $(0, 0)$. Rather than describe $P[G_2, E_2]$, we will describe the final program constructed, $P[G_2, E_4]$ (see Fig. 2).

As can be seen from the comments in the two programs, program $P[G_2, E_4]$ has the same basic structure as $P[G_2, E_1]$.

- `main` begins with initialization statements for the four example inputs.
- `Start` has five cases that correspond to the five productions of G_2 .

The main difference is that because the encoding of G_2 in `Start` uses non-determinism, we need to make sure that along *each* path p in $P[G_2, E_4]$, each of the example inputs is used to evaluate the *same* expression-tree. We address this issue by threading the expression-evaluation computations associated with each of the example inputs through the *same* non-deterministic choices. That is, each of the five “production cases” in `Start` has four encodings of the production’s semantics—one for each of the four expression evaluations. By this means, the statements that are executed along path p perform *four simultaneous bottom-up evaluations* of the expression-tree from G_2 that corresponds to p .

Programs $P[G_2, E_2]$ and $P[G_2, E_3]$ are similar to $P[G_2, E_4]$, but their paths carry out two and three simultaneous bottom-up evaluations, respectively. The

```

1 int I_0, I_1, I_2, I_3;
2 void Start(int x_0,int y_0,...,int x_3,int y_3){
3   if(nd()){ // Encodes ‘‘Start ::= Plus(Start, Start)’’
4     Start(x_0, y_0, x_1, y_1, x_2, y_2, x_3, y_3);
5     int tempL_0 = I_0; int tempL_1 = I_1;
6     int tempL_2 = I_2; int tempL_3 = I_3;
7     Start(x_0, y_0, x_1, y_1, x_2, y_2, x_3, y_3);
8     int tempR_0 = I_0; int tempR_1 = I_1;
9     int tempR_2 = I_2; int tempR_3 = I_3;
10    I_0 = tempL_0 + tempR_0;
11    I_1 = tempL_1 + tempR_1;
12    I_2 = tempL_2 + tempR_2;
13    I_3 = tempL_3 + tempR_3;}
14  else if(nd()) { // Encodes ‘‘Start ::= x’’
15    I_0 = x_0; I_1 = x_1; I_2 = x_2; I_3 = x_3;}
16  else if(nd()) { // Encodes ‘‘Start ::= y’’
17    I_0 = y_0; I_1 = y_1; I_2 = y_2; I_3 = y_3;}
18  else if(nd()) { // Encodes ‘‘Start ::= 1’’
19    I_0 = 1; I_1 = 1; I_2 = 1; I_3 = 1;}
20  else { // Encodes ‘‘Start ::= 0’’
21    I_0 = 0; I_1 = 0; I_2 = 0; I_3 = 0;}
22 }
23
24 bool spec(int x, int y, int f){
25   return (f>=x && f>=y && (f==x || f==y))
26 }
27
28 void main(){
29   int x_0 = 0; int y_0 = 1; // Input example (0,1)
30   int x_1 = 0; int y_1 = 0; // Input example (0,0)
31   int x_2 = 1; int y_2 = 1; // Input example (1,1)
32   int x_3 = 1; int y_3 = 0; // Input example (1,0)
33   Start(x_0,y_0,x_1,y_1,x_2,y_2,x_3,y_3);
34   assert( !spec(x_0,y_0,I_0) || !spec(x_1,y_1,I_1)
35         || !spec(x_2,y_2,I_2) || !spec(x_3,y_3,I_3));
36 }

```

Fig. 2. Program $P[G_2, E_4]$ created during the course of proving the unrealizability of $(\psi_{\max 2}(f, x, y), G_2)$ using the set of input examples $E_4 = \{(0, 0), (0, 1), (1, 0), (1, 1)\}$.

actions taken during rounds 2 and 3 to generate a new counter-example—and hence a new example input—are similar to what was described for round 1. On round 4, however, the program analyzer will determine that the assertion on lines (34)–(35) always holds, which means that there is no path through $P[G_2, E_4]$ for which the behavioral specification holds for all of the input examples. This property means that there is no expression-tree that satisfies the specification—i.e., the SYGUS problem $(\psi_{\max 2}(f, x, y), G_2)$ is unrealizable.

Our implementation uses the program-analysis tool SEAHORN [8] as the assertion checker. In the case of $P[G_2, E_4]$, SEAHORN takes only 0.5 s to establish that the assertion in $P[G_2, E_4]$ always holds.

3 SYGUS, Realizability, and CEGIS

3.1 Background

Trees and Tree Grammars. A *ranked alphabet* is a tuple (Σ, rk_Σ) where Σ is a finite set of symbols and $rk_\Sigma : \Sigma \rightarrow \mathbb{N}$ associates a rank to each symbol. For every $m \geq 0$, the set of all symbols in Σ with rank m is denoted by $\Sigma^{(m)}$. In our examples, a ranked alphabet is specified by showing the set Σ and attaching the respective rank to every symbol as a superscript—e.g., $\Sigma = \{+,^{(2)}, c^{(0)}\}$. (For brevity, the superscript is sometimes omitted). We use T_Σ to denote the set of all (ranked) trees over Σ —i.e., T_Σ is the smallest set such that (i) $\Sigma^{(0)} \subseteq T_\Sigma$, (ii) if $\sigma^{(k)} \in \Sigma^{(k)}$ and $t_1, \dots, t_k \in T_\Sigma$, then $\sigma^{(k)}(t_1, \dots, t_k) \in T_\Sigma$. In what follows, we assume a fixed ranked alphabet (Σ, rk_Σ) .

In this paper, we focus on *typed* regular tree grammars, in which each non-terminal and each symbol is associated with a type. There is a finite set of types $\{\tau_1, \dots, \tau_k\}$. Associated with each symbol $\sigma^{(i)} \in \Sigma^{(i)}$, there is a type assignment $a_{\sigma^{(i)}} = (\tau_0, \tau_1, \dots, \tau_i)$, where τ_0 is called the *left-hand-side type* and τ_1, \dots, τ_i are called the *right-hand-side types*. Tree grammars are similar to word grammars, but generate trees over a ranked alphabet instead of words.

Definition 1 (Regular Tree Grammar). A *typed regular tree grammar (RTG)* is a tuple $G = (N, \Sigma, S, a, \delta)$, where N is a finite set of non-terminal symbols of arity 0; Σ is a ranked alphabet; $S \in N$ is an initial non-terminal; a is a type assignment that gives types for members of $\Sigma \cup N$; and δ is a finite set of productions of the form $A_0 \rightarrow \sigma^{(i)}(A_1, \dots, A_i)$, where for $1 \leq j \leq i$, each $A_j \in N$ is a non-terminal such that if $a(\sigma^{(i)}) = (\tau_0, \tau_1, \dots, \tau_i)$ then $a(A_j) = \tau_j$.

In a SYGUS problem, each variable, such as x and y in the example RTGs in Sect. 1, is treated as an arity-0 symbol—i.e., $x^{(0)}$ and $y^{(0)}$.

Given a tree $t \in T_{\Sigma \cup N}$, applying a production $r = A \rightarrow \beta$ to t produces the tree t' resulting from replacing the left-most occurrence of A in t with the right-hand side β . A tree $t \in T_\Sigma$ is generated by the grammar G —denoted by $t \in L(G)$ —iff it can be obtained by applying a sequence of productions $r_1 \dots r_n$ to the tree whose root is the initial non-terminal S .

Syntax-Guided Synthesis. A SYGUS problem is specified with respect to a background theory T —e.g., linear arithmetic—and the goal is to synthesize a function f that satisfies two constraints provided by the user. The first constraint, $\psi(f, \bar{x})$, describes a *semantic property* that f should satisfy. The second constraint limits the *search space* S of f , and is given as a set of expressions specified by an RTG G that defines a subset of all terms in T .

Definition 2 (SYGUS). A SYGUS problem over a background theory T is a pair $sy = (\psi(f, \bar{x}), G)$ where G is a regular tree grammar that only contains terms in T —i.e., $L(G) \subseteq T$ —and $\psi(f, \bar{x})$ is a Boolean formula constraining the semantic behavior of the synthesized program f .

A SYGUS problem is **realizable** if there exists a expression $e \in L(G)$ such that $\forall \bar{x}. \psi([e], \bar{x})$ is true. Otherwise we say that the problem is **unrealizable**.

Theorem 1 (Undecidability [6]). *Given a SYGUS problem sy , it is undecidable to check whether sy is realizable.*

Counterexample-Guided Inductive Synthesis. The Counterexample-Guided Inductive Synthesis (CEGIS) algorithm is a popular approach to solving synthesis problems. Instead of directly looking for an expression that satisfies the specification φ on *all* possible inputs, the CEGIS algorithm uses a synthesizer S that can find expressions that are correct on a *finite* set of examples E . If S finds a solution that is correct on all elements of E , CEGIS uses a verifier V to check whether the discovered solution is also correct for all possible inputs to the problem. If not, a counterexample obtained from V is added to the set of examples, and the process repeats. More formally, CEGIS starts with an empty set of examples E and repeats the following steps:

1. Call the synthesizer S to find an expression e such that $\psi^E(\llbracket e \rrbracket, \bar{x}) \stackrel{\text{def}}{=} \forall \bar{x} \in E. \psi(\llbracket e \rrbracket, \bar{x})$ holds and go to step 2; return *unrealizable* if no expression exists.
2. Call the verifier V to find a model c for the formula $\neg\psi(\llbracket e \rrbracket, \bar{x})$, and add c to the counterexample set E ; return e as a valid solution if no model is found.

Because SYGUS problems are only defined over first-order decidable theories, any SMT solver can be used as the verifier V to check whether the formula $\neg\psi(\llbracket e \rrbracket, \bar{x})$ is satisfiable. On the other hand, providing a synthesizer S to find solutions such that $\forall \bar{x} \in E. \psi(\llbracket e \rrbracket, \bar{x})$ holds is a much harder problem because e is a second-order term drawn from an infinite search space. In fact, checking whether such an e exists is an undecidable problem [6].

The main contribution of our paper is a reduction of the unrealizability problem—i.e., the problem of proving that there is no expression $e \in L(G)$ such that $\forall \bar{x} \in E. \psi(\llbracket e \rrbracket, \bar{x})$ holds—to an unreachability problem (Sect. 4). This reduction allows us to use existing (un)reachability verifiers to check whether a SYGUS instance is unrealizable.

3.2 CEGIS and Unrealizability

The CEGIS algorithm is sound but incomplete for proving unrealizability. Given a SYGUS problem $sy = (\psi(f, \bar{x}), G)$ and a finite set of inputs E , we denote with $sy^E := (\psi^E(f, \bar{x}), G)$ the corresponding SYGUS problem that only requires the function f to be correct on the examples in E .

Lemma 1 (Soundness). *If sy^E is unrealizable then sy is unrealizable.*

Even when given a perfect synthesizer S —i.e., one that can solve a problem sy^E for every possible set E —there are SYGUS problems for which the CEGIS algorithm is not powerful enough to prove unrealizability.

Lemma 2 (Incompleteness). *There exists an unrealizable SYGUS problem sy such that for every finite set of examples E the problem sy^E is realizable.*

Despite this negative result, we will show that a CEGIS algorithm can prove unrealizability for many SYGUS instances (Sect. 5).

4 From Unrealizability to Unreachability

In this section, we show how a SYGUS problem for finitely many examples can be reduced to a reachability problem in a non-deterministic, recursive program in an imperative programming language.

4.1 Reachability Problems

A program P takes an initial state I as input and outputs a final state O , i.e., $\llbracket P \rrbracket(I) = O$ where $\llbracket \cdot \rrbracket$ denotes the semantic function of the programming language. As illustrated in Sect. 2, we allow a program to contain calls to an external function `nd()`, which returns a non-deterministically chosen Boolean value. When program P contains calls to `nd()`, we use \hat{P} to denote the program that is the same as P except that \hat{P} takes an additional integer input `n`, and each call `nd()` is replaced by a call to a local function `nextbit()` defined as follows:

```
bool nextbit(){bool b = n%2; n=n>>1; return b;}.
```

In other words, the integer parameter `n` of $\hat{P}[n]$ formalizes all of the non-deterministic choices made by P in calls to `nd()`.

For the programs $P[G, E]$ used in our unrealizability algorithm, the only calls to `nd()` are ones that control whether or not a production is selected from grammar G during a top-down, left-to-right generation of an expression-tree. Given `n`, we can decode it to identify which expression-tree `n` represents.

Example 1. Consider again the SYGUS problem $(\psi_{\max 2}(f, x, y), G_2)$ discussed in Sect. 2. In the discussion of the initial program $P[G_2, E_1]$ (Fig. 1), we hypothesized that the program analyzer chose to report path (1) in P , for which the sequence of non-deterministic choices is t, f, t, f, f, f, t . That sequence means that for $\hat{P}[n]$, the value of `n` is 1000101 (base 2) (or 69 (base 10)). The 1s, from low-order to high-order position, represent choices of production instances in a top-down, left-to-right generation of an expression-tree. (The 0s represent rejected possible choices). The rightmost 1 in `n` corresponds to the choice in line (3) of “`Start ::= Plus(Start, Start)`”; the 1 in the third-from-rightmost position corresponds to the choice in line (10) of “`Start ::= x`” as the left child of the `Plus` node; and the 1 in the leftmost position corresponds to the choice in line (12) of “`Start ::= 1`” as the right child. By this means, we learn that the behavioral specification $\psi_{\max 2}(f, x, y)$ holds for the example set $E_1 = \{(0, 1)\}$ for $f \mapsto \text{Plus}(x, 1)$. \square

Definition 3 (Reachability Problem). *Given a program $\hat{P}[n]$, containing assertion statements and a non-deterministic integer input `n`, we use rep_P to denote the corresponding reachability problem. The reachability problem rep_P is **satisfiable** if there exists a value n that, when bound to `n`, falsifies any of the assertions in $\hat{P}[n]$. The problem is **unsatisfiable** otherwise.*

4.2 Reduction to Reachability

The main component of our framework is an encoding enc that given a SYGUS problem $sy^E = (\psi^E(f, x), G)$ over a set of examples $E = \{c_1, \dots, c_k\}$, outputs a program $P[G, E]$ such that sy^E is **realizable** if and only if $re_{enc(sy, E)}$ is **satisfiable**. In this section, we define all the components of $P[G, E]$, and state the correctness properties of our reduction.

Remark: In this section, we assume that in the specification $\psi(f, x)$ every occurrence of f has x as input parameter. We show how to overcome this restriction in App. A [13]. In the following, we assume that the input x has type τ_I , where τ_I could be a complex type—e.g., a tuple type.

Program Construction. Recall that the grammar G is a tuple $(N, \Sigma, S, a, \delta)$. First, for each non-terminal $A \in N$, the program $P[G, E]$ contains k global variables $\{g_1_A, \dots, g_k_A\}$ of type $a(A)$ that are used to express the values resulting from evaluating expressions generated from non-terminal A on the k examples. Second, for each non-terminal $A \in N$, the program $P[G, E]$ contains a function

```
void funcA( $\tau_I$  v1, ...,  $\tau_I$  vk) { bodyA }
```

We denote by $\delta(A) = \{r_1, \dots, r_m\}$ the set of production rules of the form $A \rightarrow \beta$ in δ . The body $bodyA$ of $funcA$ has the following structure:

```
if(nd()) {Enδ(r1)}
else if(nd()) {Enδ(r2)}
...
else {Enδ(rm)}
```

The encoding $En_{\delta}(r)$ of a production $r = A_0 \rightarrow b^{(j)}(A_1, \dots, A_j)$ is defined as follows (τ_i denotes the type of the term A_i):

```
funcA1(v1, ..., vk);
 $\tau_1$  child_1_1 = g_1_A1; ...;  $\tau_1$  child_1_k = g_k_Aj;
...
funcAj(v1, ..., vk);
 $\tau_j$  child_j_1 = g_1_A1; ...;  $\tau_j$  child_j_k = g_k_Aj;
g_1_A0 = encb1(child_1_1, ..., child_1_k)
...
g_k_A0 = encbk(child_j_1, ..., child_j_k)
```

Note that if $b^{(j)}$ is of arity 0—i.e., if $j = 0$ —the construction yields k assignments of the form $g_m A0 = enc_b^m()$.

The function enc_b^m interprets the semantics of b on the m^{th} input example. We take Linear Integer Arithmetic as an example to illustrate how enc_b^m works.

$$\begin{array}{ll} enc_{0(0)}^m := 0 & enc_{1(0)}^m := 1 \\ enc_{x(0)}^m := vi & enc_{\text{Equals}(2)}^m(L, R) := (L=R) \\ enc_{\text{Plus}(2)}^m(L, R) := L+R & enc_{\text{Minus}(2)}^m(L, R) := L-R \\ enc_{\text{IfThenElse}(3)}^m(B, L, R) := \text{if}(B) L \text{ else } R & \end{array}$$

We now turn to the correctness of the construction. First, we formalize the relationship between expression-trees in $L(G)$, the semantics of $P[G, E]$, and the number \mathbf{n} . Given an expression-tree e , we assume that each node q in e is annotated with the production that has produced that node. Recall that $\delta(A) = \{r_1, \dots, r_m\}$ is the set of productions with head A (where the subscripts are indexes in some arbitrary, but fixed order). Concretely, for every node q , we assume there is a function $pr(q) = (A, i)$, which associates q with a pair that indicates that non-terminal A produced n using the production r_i (i.e., r_i is the i^{th} production whose left-hand-side non-terminal is A).

We now define how we can extract a number $\#(e)$ for which the program $\hat{P}[\#(e)]$ will exhibit the same semantics as that of the expression-tree e . First, for every node q in e such that $pr(q) = (A, i)$, we define the following number:

$$\#_{nd}(q) = \begin{cases} \underbrace{1 0 \cdots 0}_{i-1} & \text{if } i < |\delta(A)| \\ \underbrace{0 \cdots 0}_{i-1} & \text{if } i = |\delta(A)|. \end{cases}$$

The number $\#_{nd}(q)$ indicates what suffix of the value of \mathbf{n} will cause `funcA` to trigger the code corresponding to production r_i . Let $q_1 \cdots q_m$ be the sequence of nodes visited during a pre-order traversal of expression-tree e . The number corresponding to e , denoted by $\#(e)$, is defined as the bit-vector $\#_{nd}(q_m) \cdots \#_{nd}(q_1)$.

Finally, we add the entry-point of the program, which calls the function `funcS` corresponding to the initial non-terminal S , and contains the assertion that encodes our reachability problem on all the input examples $E = \{c_1, \dots, c_k\}$.

```
void main(){
     $\tau_I$  x1 = c1;  $\cdots$ ;  $\tau_I$  xk = ck;
    funcS(x1,  $\dots$ , xk);
    assert  $\bigvee_{1 \leq i \leq k} \neg \psi(f, c_i)[g\_i\_S/f(x)]$ ; // At least one  $c_i$  fails }
```

Correctness. We first need to show that the function $\#(\cdot)$ captures the correct language of expression-trees. Given a non-terminal A , a value n , and input values i_1, \dots, i_k , we use $\llbracket \text{funcA}[n] \rrbracket(i_1, \dots, i_k) = (o_1, \dots, o_k)$ to denote the values of the variables $\{g_1_A, \dots, g_k_A\}$ at the end of the execution of `funcA[n]` with the initial value of $\mathbf{n} = n$ and input values x_1, \dots, x_k . Given a non-terminal A , we write $L(G, A)$ to denote the set of terms that can be derived starting with A .

Lemma 3. *Let A be a non-terminal, $e \in L(G, A)$ an expression, and $\{i_1, \dots, i_k\}$ an input set. Then, $(\llbracket e \rrbracket(i_1), \dots, \llbracket e \rrbracket(i_k)) = \llbracket \text{funcA}[\#(e)] \rrbracket(i_1, \dots, i_k)$.*

Each procedure `funcA[n](i1, ..., ik)` that we construct has an explicit dependence on variable \mathbf{n} , where \mathbf{n} controls the non-deterministic choices made by the `funcA` and procedures called by `funcA`. As a consequence, when relating numbers and expression-trees, there are two additional issues to contend with:

Non-termination. Some numbers can cause `funcA[n]` to fail to terminate.

For instance, if the case for “Start ::= Plus(Start, Start)” in program

$P[G_2, E_1]$ from Fig. 1 were moved from the first branch (lines (3)–(8)) to the final else case (line (13)), the number $n = 0 = \dots 0000000$ (base 2) would cause **Start** to never terminate, due to repeated selections of **Plus** nodes. However, note that the only assert statement in the program is placed at the end of the main procedure. Now, consider a value of n such that $re_{enc(sy, E)}$ is satisfiable. Definition 3 implies that the flow of control will reach and falsify the assertion, which implies that $\text{funcA}[n]$ terminates.⁴

Shared suffixes of sufficient length. In Example 1, we showed how for program $P[G_2, E_1]$ (Fig. 1) the number $n = 1000101$ (base 2) corresponds to the top-down, left-to-right generation of $\text{Plus}(x, 1)$. That derivation consumed exactly seven bits; thus, any number that, written in base 2, shares the suffix 1000101 —e.g., 11010101000101 —will also generate $\text{Plus}(x, 1)$.

The issue of shared suffixes is addressed in the following lemma:

Lemma 4. *For every non-terminal A and number n such that $[\![\text{funcA}[n]]\!](i_1, \dots, i_k) \neq \perp$ (i.e., funcA terminates when the non-deterministic choices are controlled by n), there exists a minimal n' that is a (base 2) suffix of n for which (i) there is an $e \in L(G)$ such that $\#(e) = n'$, and (ii) for every input $\{i_1, \dots, i_k\}$, we have $[\![\text{funcA}[n]]\!](i_1, \dots, i_k) = [\![\text{funcA}[n']]\!](i_1, \dots, i_k)$.*

We are now ready to state the correctness property of our construction.

Theorem 2. *Given a SYGUS problem $sy^E = (\psi_E(f, x), G)$ over a finite set of examples E , the problem sy^E is **realizable** iff $re_{enc(sy, E)}$ is **satisfiable**.*

5 Implementation and Evaluation

NOPE is a tool that can return two-sided answers to unrealizability problems of the form $sy = (\psi, G)$. When it returns **unrealizable**, no expression-tree in $L(G)$ satisfies ψ ; when it returns **realizable**, some $e \in L(G)$ satisfies ψ ; NOPE can also time out. NOPE incorporates several existing pieces of software.

1. The (un)reachability verifier SEAHORN is applied to the reachability problems of the form $re_{enc(sy, E)}$ created during successive CEGIS rounds.
2. The SMT solver Z3 is used to check whether a generated expression-tree e satisfies ψ . If it does, NOPE returns **realizable** (along with e); if it does not, NOPE creates a new input example to add to E .

It is important to observe that SEAHORN, like most reachability verifiers, is only sound for **unsatisfiability**—i.e., if SEAHORN returns **unsatisfiable**, the reachability problem is indeed unsatisfiable. Fortunately, SEAHORN’s one-sided

⁴ If the SYGUS problem deals with the synthesis of programs for a language that can express non-terminating programs, that would be an additional source of non-termination, different from that discussed in item **Non-termination**. That issue does not arise for LIA SYGUS problems. Dealing with the more general kind of non-termination is postponed for future work.

answers are in the correct direction for our application: to prove unrealizability, NOPE only requires the reachability verifier to be sound for unsatisfiability.

There is one aspect of NOPE that differs from the technique that has been presented earlier in the paper. While SEAHORN is sound for *unreachability*, it is not sound for reachability—i.e., it cannot soundly prove whether a synthesis problem is realizable. To address this problem, to check whether a given SYGUS problem sy^E is realizable on the finite set of examples E , NOPE also calls the SYGUS solver ESolver [2] to synthesize an expression-tree e that satisfies sy^E .⁵

In practice, for every intermediate problem sy^E generated by the CEGIS algorithm, NOPE runs the ESolver on sy^E and SEAHORN on $re_{enc}(sy, E)$ in *parallel*. If ESolver returns a solution e , SEAHORN is interrupted, and Z3 is used to check whether e satisfies ψ . Depending on the outcome, NOPE either returns *realizable* or obtains an additional input example to add to E . If SEAHORN returns *unsatisfiable*, NOPE returns *unrealizable*.

Modulo bugs in its constituent components, NOPE is sound for both realizability and unrealizability, but because of Lemma 2 and the incompleteness of SEAHORN, NOPE is not complete for unrealizability.

Benchmarks. We perform our evaluation on 132 variants of the 60 LIA benchmarks from the LIA SYGUS competition track [2]. We do not consider the other SYGUS benchmark track, Bit-Vectors, because the SEAHORN verifier is unsound for most bit-vector operations—e.g., bit-shifting. We used three suites of benchmarks. LIMITEDIF (resp. LIMITEDPLUS) contains 57 (resp. 30) benchmarks in which the grammar bounds the number of times an IfThenElse (resp. Plus) operator can appear in an expression-tree to be 1 less than the number required to solve the original synthesis problem. We used the tool QUASI to automatically generate the restricted grammars. LIMITEDCONST contains 45 benchmarks in which the grammar allows the program to contain only constants that are coprime to any constants that may appear in a valid solution—e.g., the solution requires using odd numbers, but the grammar only contains the constant 2. The numbers of benchmarks in the three suites differ because for certain benchmarks it did not make sense to create a limited variant—e.g., if the smallest program consistent with the specification contains no IfThenElse operators, no variant is created for the LIMITEDIF benchmark. In all our benchmarks, the grammars describing the search space contain infinitely many terms.

Our experiments were performed on an Intel Core i7 4.00 GHz CPU, with 32 GB of RAM, running Lubuntu 18.10 via VirtualBox. We used version 4.8 of Z3, commit 97f2334 of SEAHORN, and commit d37c50e of ESolver. The timeout for each individual SEAHORN/ESolver call is set at 10 min.

Experimental Questions. Our experiments were designed to answer the questions posed below.

EQ 1. Can NOPE prove unrealizability for variants of real SYGUS benchmarks, and how long does it take to do so?

⁵ We chose ESolver because on the benchmarks we considered, ESolver outperformed other SYGUS solvers (e.g., CVC4 [3]).

Finding: NOPE can prove unrealizability for 59/132 benchmarks. For the 59 benchmarks solved by NOPE, the average time taken is 15.59 s. The time taken to perform the last iteration of the algorithm—i.e., the time taken by SEAHORN to return **unsatisfiable**—accounts for 87% of the total running time.

NOPE can solve all of the LIMITEDIF benchmarks for which the grammar allows at most one IfThenElse operator. Allowing more IfThenElse operators in the grammar leads to larger programs and larger sets of examples, and consequently the resulting reachability problems are harder to solve for SEAHORN.

For a similar reason, NOPE can solve only one of the LIMITEDPLUS benchmarks. All other LIMITEDPLUS benchmarks allow 5 or more Plus statements, resulting in grammars that have at least 130 productions.

NOPE can solve all LIMITEDCONST benchmarks because these require few examples and result in small encoded programs.

EQ 2. How many examples does NOPE use to prove unrealizability and how does the number of examples affect the performance of NOPE?

Note that Z3 can produce different models for the same query, and thus different runs of NOPE can produce different sequences of examples. Hence, there is no guarantee that NOPE finds a good sequence of examples that prove unrealizability. One measure of success is whether NOPE is generally able to find a small number of examples, when it succeeds in proving unrealizability.

Finding: Nope used 1 to 9 examples to prove unrealizability for the benchmarks on which it terminated. Problems requiring large numbers of examples could not be solved because either ESolver or times out—e.g., on the problem max4, NOPE gets to the point where the CEGIS loop has generated 17 examples, at which point ESolver exceeds the timeout threshold.

Finding: The number of examples required to prove unrealizability depends mainly on the arity of the synthesized function and the complexity of the grammar. The number of examples seems to grow quadratically with the number of bounded operators allowed in the grammar. In particular, problems in which the grammar allows zero IfThenElse operators require 2–4 examples, while problems in which the grammar allows one IfThenElse operator require 7–9 examples.

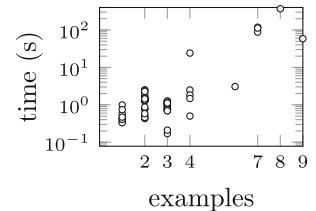


Fig. 3. Time vs examples.

Figure 3 plots the running time of NOPE against the number of examples generated by the CEGIS algorithm. *Finding:* The solving time appears to grow exponentially with the number of examples required to prove unrealizability.

6 Related Work

The SYGUS formalism was introduced as a unifying framework to express several synthesis problems [1]. Caulfield et al. [6] proved that it is undecidable to determine whether a given SYGUS problem is realizable. Despite this negative result,

there are several SYGUS solvers that compete in yearly SYGUS competitions [2] and can efficiently produce solutions to SYGUS problems when a solution exists. Existing SYGUS synthesizers fall into three categories: (i) Enumeration solvers enumerate programs with respect to a given total order [7]. If the given problem is unrealizable, these solvers typically only terminate if the language of the grammar is finite or contains finitely many functionally distinct programs. While in principle certain enumeration solvers can prune infinite portions of the search space, none of these solvers could prove unrealizability for any of the benchmarks considered in this paper. (ii) Symbolic solvers reduce the synthesis problem to a constraint-solving problem [3]. These solvers cannot reason about grammars that restrict allowed terms, and resort to enumeration whenever the candidate solution produced by the constraint solver is not in the restricted search space. Hence, they also cannot prove unrealizability. (iii) Probabilistic synthesizers randomly search the search space, and are typically unpredictable [14], providing no guarantees in terms of unrealizability.

Synthesis as Reachability. CETI [12] introduces a technique for encoding template-based synthesis problems as reachability problems. The CETI encoding only applies to the specific setting in which (i) the search space is described by an imperative program with a *finite number* of holes—i.e., the values that the synthesizer has to discover—and (ii) the specification is given as a finite number of input-output test cases with which the target program should agree. Because the number of holes is finite, and all holes correspond to values (and not terms), the reduction to a reachability problem only involves making the holes global variables in the program (and no more elaborate transformations).

In contrast, our reduction technique handles search spaces that are described by a grammar, which in general consist of an *infinite* set of terms (not just values). Due to this added complexity, our encoding has to account for (i) the semantics of the productions in the grammar, and (ii) the use of non-determinism to encode the choice of grammar productions. Our encoding creates one expression-evaluation computation for each of the example inputs, and threads these computations through the program so that each expression-evaluation computation makes use of the *same* set of non-deterministic choices.

Using the input-threading, our technique can handle specifications that contain nested calls of the synthesized program (e.g., $f(f(x)) = x$). (App. A [13]).

The input-threading technique builds a *product program* that performs multiple executions of the same function as done in relational program verification [4]. Alternatively, a different encoding could use multiple function invocations on individual inputs and require the verifier to thread the same bit-stream for all input evaluations. In general, verifiers perform much better on product programs [4], which motivates our choice of encoding.

Unrealizability in Program Synthesis. For certain synthesis problems—e.g., reactive synthesis [5]—the realizability problem is decidable. The framework tackled in this paper, SYGUS, is orthogonal to such problems, and it is undecidable to check whether a given SYGUS problem is realizable [6].

Mechtaev et al. [11] propose to use a variant of SYGUS to efficiently prune irrelevant paths in a symbolic-execution engine. In their approach, for each path π in the program, a synthesis problem p_π is generated so that if p_π is unrealizable, the path π is infeasible. The synthesis problems generated by Mechtaev et al. (which are not directly expressible in SYGUS) are decidable because the search space is defined by a finite set of templates, and the synthesis problem can be encoded by an SMT formula. To the best of our knowledge, our technique is the first one that can check unrealizability of general SYGUS problems in which the search space is an *infinite set of functionally distinct terms*.

Acknowledgment. This work was supported, in part, by a gift from Rajiv and Ritu Batra; by AFRL under DARPA MUSE award FA8750-14-2-0270 and DARPA STAC award FA8750-15-C-0082; by ONR under grant N00014-17-1-2889; by NSF under grants CNS-1763871 and CCF-1704117; and by the UW-Madison OVRGE with funding from WARF.

References

1. Alur, R., et al.: Syntax-guided synthesis. In: Formal Methods in Computer-Aided Design (FMCAD), pp. 1–8. IEEE (2013)
2. Alur, R., Fisman, D., Singh, R., Solar-Lezama, A.: SyGuS-Comp 2016: results and analysis. arXiv preprint [arXiv:1611.07627](https://arxiv.org/abs/1611.07627) (2016)
3. Barrett, C., et al.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_14
4. Barthe, G., Crespo, J.M., Kunz, C.: Relational verification using product programs. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 200–214. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21437-0_17
5. Bloem, R.: Reactive synthesis. In: Formal Methods in Computer-Aided Design (FMCAD), p. 3 (2015)
6. Caulfield, B., Rabe, M.N., Seshia, S.A., Tripakis, S.: What’s decidable about syntax-guided synthesis? arXiv preprint [arXiv:1510.08393](https://arxiv.org/abs/1510.08393) (2015)
7. ESolver. <https://github.com/abhishekudupa/sygus-comp14>
8. Gurfinkel, A., Kahsai, T., Navas, J.A.: SeaHorn: a framework for verifying C programs (competition contribution). In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 447–450. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_41
9. Hu, Q., D’Antoni, L.: Automatic program inversion using symbolic transducers. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pp. 376–389 (2017)
10. Hu, Q., D’Antoni, L.: Syntax-guided synthesis with quantitative syntactic objectives. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 386–403. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_21
11. Mechtaev, S., Griggio, A., Cimatti, A., Roychoudhury, A.: Symbolic execution with existential second-order constraints. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), pp. 389–399 (2018)

12. Nguyen, T.V., Weimer, W., Kapur, D., Forrest, S.: Connecting program synthesis and reachability: automatic program repair using test-input generation. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10205, pp. 301–318. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_17
13. Qinheping, H., Jason, B., John, C., Loris, D., Reps, T.: Proving unrealizability for syntax-guided synthesis. arXiv preprint [arXiv:1905.05800](https://arxiv.org/abs/1905.05800) (2019)
14. Schkufza, E., Sharma, R., Aiken, A.: Stochastic program optimization. Commun. ACM **59**(2), 114–122 (2016)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.



Model Checking



BMC for Weak Memory Models: Relation Analysis for Compact SMT Encodings

Natalia Gavrilenko^{1,4(✉)}, Hernán Ponce-de-León²,
Florian Furbach³, Keijo Heljanko⁴,
and Roland Meyer³

¹ Aalto University, Helsinki, Finland

² fortiss GmbH, Munich, Germany

³ TU Braunschweig, Brunswick, Germany

⁴ University of Helsinki and HIIT, Helsinki, Finland

natalia.gavrilenko@helsinki.fi



Abstract. We present DARTAGNAN, a bounded model checker (BMC) for concurrent programs under weak memory models. Its distinguishing feature is that the memory model is not implemented inside the tool but taken as part of the input. DARTAGNAN reads CAT, the standard language for memory models, which allows to define x86/TSO, ARMv7, ARMv8, POWER, C/C++, and LINUX kernel concurrency primitives. BMC with memory models as inputs is challenging. One has to encode into SMT not only the program but also its semantics as defined by the memory model. What makes DARTAGNAN scale is its relation analysis, a novel static analysis that significantly reduces the size of the encoding. DARTAGNAN matches or even exceeds the performance of the model-specific verification tools NIDHUGG and CBMC, as well as the performance of HERD, a CAT-compatible litmus testing tool. Compared to the unoptimized encoding, the speed-up is often more than two orders of magnitude.

Keywords: Weak memory models · CAT · Concurrency · BMC · SMT

1 Introduction

When developing concurrency libraries or operating system kernels, performance and scalability of the concurrency primitives is of paramount importance. These primitives rely on the synchronization guarantees of the underlying hardware and the programming language runtime environment. The formal semantics of these guarantees are often defined in terms of weak memory models. There is considerable interest in verification tools that take memory models into account [5, 9, 13, 22].

A successful approach to formalizing weak memory models is CAT [11, 12, 16], a flexible specification language in which all memory models considered so far can be expressed succinctly. CAT, together with its accompanying tool HERD [4],

has been used to formalize the semantics not only of assembly for x86/TSO, POWER, ARMv7 and ARMv8, but also high-level programming languages, such as C/C++, transactional memory extensions, and recently the LINUX kernel concurrency primitives [11, 15, 16, 18, 20, 24, 29]. This success indicates the need for universal verification tools that are not limited to a specific memory model.

We present DARTAGNAN [3], a bounded model checker that takes memory models as inputs. DARTAGNAN expects a concurrent program annotated with an assertion and a memory model for which the verification should be conducted. It verifies the assertion on those executions of the program that are valid under the given memory model and returns a counterexample execution if the verification fails. As is typical of BMC, the verification results hold relative to an unrolling bound [21]. The encoding phase, however, is new. Not only the program but also its semantics as defined by the CAT model are translated into an SMT formula.

Having to take into account the semantics quickly leads to large encodings. To overcome this problem, DARTAGNAN implements a novel *relation analysis*, which can be understood as a static analysis of the program semantics as defined by the memory model. More precisely, CAT defines the program semantics in terms of relations between the events that may occur in an execution. Depending on constraints over these relations, an execution is considered valid or invalid. Relation analysis determines the pairs of events that may influence a constraint of the memory model. Any remaining pair can be dropped from the encoding. The analysis is compatible with optimized fixpoint encodings presented in [27, 28].

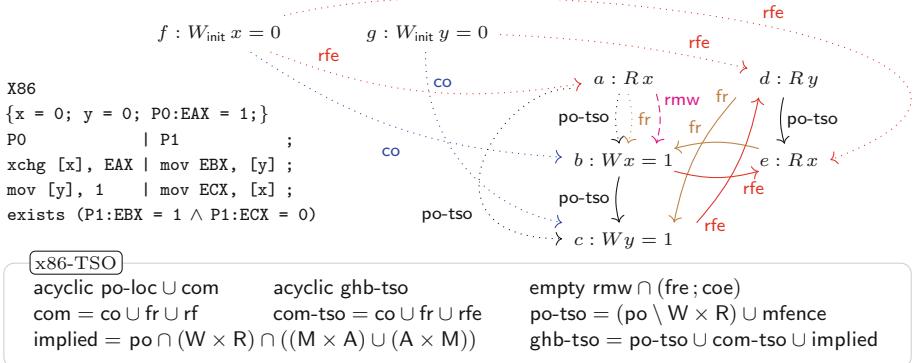
The second novelty is the support for advanced programming constructs. We redesigned DARTAGNAN’s heap model, which now has pointers and arrays. Furthermore, we enriched the set of synchronization primitives, including read-modify-write and read-copy-update (RCU) instructions [26]. One motivation for this richer set of programming constructs is the Linux kernel memory model [15] that has recently been added to the kernel documentation [2]. This model has already been used by kernel developers to find bugs in and clarify details of the concurrency primitives. Since the model is expected to be refined with further development of the kernel, verification tools will need to quickly accommodate updates in the specification. So far, only HERD [4] has satisfied this requirement. Unfortunately, it is limited to fairly small programs (litmus tests). The present version of DARTAGNAN offers an alternative with substantially better performance.

We present experiments on a series of benchmarks consisting of 4751 LINUX litmus tests and 7 mutual exclusion algorithms executed on TSO, ARM, and LINUX. Despite the flexibility of taking memory models as inputs, DARTAGNAN’s performance is comparable to CBMC [13] and considerably better than that of NIDHUGG [5, 9]. Both are model-specific tools. Compared to the previous version of DARTAGNAN [28] and compared to HERD [4], we gain a speed-up of more than two orders of magnitude, thanks to the relation analysis.

Related Work. In terms of the verification task to be solved, the following tools are the closest to ours. CBMC [13] is a scalable bounded model checker supporting TSO, but not ARM. An earlier version also supported POWER.

NIDHUGG [5,9] is a stateless model checker supporting TSO, POWER, and a subset of ARMv7. It is excellent for programs with a small number of executions. RCMC [22] implements a stateless model checking algorithm targeting C11. We cannot directly benchmark against it because the source code of the tool is not yet publicly available, nor do we fully support C11. HERD [4] is the only tool aside from ours that takes a CAT memory model as input. HERD does not scale well to programs with a large number of executions, including some of the LINUX kernel tests. Other verification tasks (e.g., fence insertion to restore sequential consistency) are tackled by MEMORAX [6–8], OFFENCE [14], FENDER [23], DFENCE [25], and TRENCHER [19].

Relation Analysis on an Example. Consider the program (in the .litmus format) given to the left in the figure below. The assertion asks whether there is a reachable state with final values $EBX = 1, ECX = 0$. We analyze the program under the x86-TSO memory model shown below the program. The semantics of the program under TSO is a set of executions. An execution is a graph, similar to the one given below, where the nodes are events and the edges correspond to the relations defined by the memory model. Events are instances of instructions that access the shared memory: R (loads), W (stores, including initial stores), and M (the union of both). The atomic exchange instruction $xchq [x], EAX$ gives rise to a pair of read and write events related by a (dashed) rmw edge. Such reads and writes belong to the set A of atomic read-modify-write events.



The relations rf , co , and fr model the communication of instructions via the shared memory (reading from a write, coherence, overwriting a read). Their restrictions rfe , coe , and fre denote (external) communication between instructions from different threads. Relation po is the program order within the same thread and $po\text{-loc}$ is its restriction to events addressing the same memory location. Edges of $mfence$ relate events separated by a fence. Further relations are derived from these base relations. To belong to the TSO semantics of the program, an execution has to satisfy the constraints of the memory model: $empty\ rmw \cap (fre\ ;\ coe)$, which enforces atomicity of read-modify-write events, and the two acyclicity constraints.

DARTAGNAN encodes the semantics of the given program under the given memory model into an SMT formula. The problem is that each edge (a, b) that may be present in a relation r gives rise to a variable $r(a, b)$. The goal of our relation analysis is to reduce the number of edges that need to be encoded. We illustrate this on the constraint `acyclic ghb-tso`. The graph next to the program shows the 14 (dotted and solid) edges which may contribute to the relation `ghb-tso`. Of those, only the 6 solid edges can occur in a cycle. The dotted edges can be dropped from the SMT encoding. Our relation analysis determines the solid edges—edges that may have an influence on a constraint of the memory model. Additionally, `ghb-tso` is a composition of various subrelations (e.g., `po-tso` or `co ∪ fr`) that also require encoding into SMT. Relation analysis applies to subrelations as well. Applied to all constraints, it reduces the number of encoded edges for all (sub)relations from 221 to 58.

2 Input, Functionality, and Implementation

DARTAGNAN has the ambition of being widely applicable, from assembly over operating system code written in C/C++ to lock-free data structures. The tool accepts programs in PPC, x86, AArch64 assembly, and a subset of C11, all limited to the subsets supported by Herd’s .litmus format. It also reads our own .pts format with C11-like syntax [28]. We refer to global variables as memory locations and to local variables as registers. We support pointers, i.e., a register may hold the address of a location. Addresses and values are integers, and we allow the same arithmetic operations for addresses as for regular integer values. Different synchronization mechanisms are available, including variants of read-modify-write, various fences, and RCU instructions [26].

We support the assertion language of HERD. Assertions define inequalities over the values of registers and locations. They come with quantifiers over the reachable states that should satisfy the inequalities.

We use the CAT language [11, 12, 16] to define memory models. A memory model consists of named relations between events that may occur in an execution. Whether or not an execution is valid is defined by constraints over these relations:

$$\begin{aligned}
 \langle MM \rangle &::= \langle const \rangle \mid \langle rel \rangle \mid \langle MM \rangle \wedge \langle MM \rangle & \langle r \rangle &::= \langle b \rangle \mid \langle name \rangle \mid \langle r \rangle \cup \langle r \rangle \mid \langle r \rangle \setminus \langle r \rangle \\
 \langle const \rangle &::= acyclic(\langle r \rangle) \mid irreflexive(\langle r \rangle) & \mid \langle r \rangle \cap \langle r \rangle \mid \langle r \rangle^{-1} \mid \langle r \rangle^+ \mid \langle r \rangle^* \mid \langle r \rangle; \langle r \rangle \\
 &\mid empty(\langle r \rangle) & \langle b \rangle &::= id \mid int \mid ext \mid po \mid \text{fencerel}(fence) \\
 \langle rel \rangle &::= \langle name \rangle := \langle r \rangle & \mid rmw \mid ctrl \mid data \mid addr \mid loc \mid rf \mid co.
 \end{aligned}$$

CAT has a rich relational language, and we only show an excerpt above. So-called base relations $\langle b \rangle$ model the control flow, data flow, and synchronization constraints. The language provides intuitive operators to derive further relations. One may define relations recursively by referencing named relations. Their semantics is the least fixpoint.

DARTAGNAN is invoked with two inputs: the program, annotated with an assertion over the final states, and the memory model. There are two optional parameters related to the verification. The SMT encoding technique for recursive relations is defined by `mode` chosen between `knastertarski` (default) and `idl` (see below). The parameter `alias`, chosen between `none` and `andersen` (default), defines whether to use an alias analysis for our relation analysis (cf. Sect. 3).

Being a bounded model checker, DARTAGNAN computes an unrolled program with conditionals but no loops. It encodes this acyclic program together with the memory model into an SMT formula and passes it to the Z3 solver. The formula has the form $\psi_{\text{prog}} \wedge \psi_{\text{assert}} \wedge \psi_{\text{mm}}$, where ψ_{prog} encodes the program, ψ_{assert} the assertion, and ψ_{mm} the memory model. We elaborate on the encoding of the program and the memory model. The assertion is already given as a formula.

We model the heap by encoding a new memory location for each variable and a set of locations for each memory allocation of an array. Every location has an address encoded as an integer variable whose value is chosen by the solver. In an array, the locations are required to have consecutive addresses. Instances of instructions are modeled as events, most notably stores (to the shared memory) and loads (from the shared memory).

We encode relations by associating pairs of events with Boolean variables. Whether the pair (e_1, e_2) is contained in relation r is indicated by the variable $r(e_1, e_2)$. Encoding the relations $r_1 \cap r_2$, $r_1 \cup r_2$, $r_1 ; r_2$, $r_1 \setminus r_2$ and r^{-1} is straightforward [27]. For recursively defined and (reflexive and) transitive relations, DARTAGNAN lets the user choose between two methods for computing fixed points by setting the appropriate parameter. The integer-difference logic (IDL) method encodes a Kleene iteration by means of integer variables (one for each pair of events) representing the step in which the pair was added to the relation [27]. The Knaster-Tarski encoding simply looks for a post fixpoint. We have shown in [28] that this is sufficient for reachability analysis.

3 Relation Analysis

To optimize the size of the encoding (and the solving times), we found it essential to reduce the domains of the relations. We determine for each relation a static over-approximation of the pairs of events that may be in this relation. Even more, we restrict the relation to the set of pairs that may influence a constraint of the given memory model. These restricted sets are the *relation analysis* information (of the program relative to the memory model). Technically, we compute, for each relation r , two sets of event pairs, $M(r)$ and $A(r)$. The former contains so-called *may pairs*, pairs of events that may be in relation r . This does not yet take into account whether the may pairs occur in some constraint of the memory model. The *active pairs* $A(r)$ incorporate this information, and hence restrict the set of may pairs. As a consequence of the relation analysis, we only introduce Boolean variables $r(e_1, e_2)$ for the pairs $(e_1, e_2) \in A(r)$ to the SMT encoding.

The algorithm for constructing the may set and the active set is a fix-point computation. What is unconventional is that the two sets propagate their

information in different directions. For $A(r)$, the computation proceeds from the constraints and propagates information down the syntax tree of the CAT memory model. The sets $M(r)$ are computed bottom-up the syntax tree. Interestingly, in our implementation, we do not compute the full fixpoint but let the top-down process trigger the required bottom-up computation.

Both sets are computed as least solutions to a common system of inequalities. As we work over powerset lattices (relations are sets after all), the order of the system will be inclusion. We understand each set $M(r)$ and $A(r)$ as a variable, thereby identifying it with its least solution. To begin with, we give the definition for $A(r)$. In the base case, we have a relation r that occurs in a constraint of the memory model. The inequality is defined based on the shape of the constraint:

$$A(r) \supseteq M(r) \text{ (empty)} \quad A(r) \supseteq M(r) \cap \text{id} \text{ (irrefl.)} \quad A(r) \supseteq M(r) \cap M(r^+)^{-1} \text{ (acyclic).}$$

For the emptiness constraint, all pairs of events that may be contained in the relation are relevant. If the constraint requires irreflexivity, what matters are the pairs (e, e) . If the constraint requires acyclicity, we concentrate on the pairs (e_1, e_2) , where (e_1, e_2) may be in relation r and (e_2, e_1) may be in relation r^+ . Note how the definition of active pairs triggers the computation of may pairs.

If the relation in the constraint is a composed one, the following inequalities propagate the information about the active pairs down the syntax tree of the CAT memory model:

$$\begin{aligned} A(r_1) &\supseteq A(r)^{-1} && \text{if } r = r_1^{-1} \\ A(r_1) &\supseteq A(r) && \text{if } r = r_1 \cap r_2 \text{ or } r = r_1 \setminus r_2 \\ A(r_1) &\supseteq A(r) \cap M(r_1) && \text{if } r = r_1 \cup r_2 \text{ or } r = r_2 \setminus r_1 \\ A(r_1) &\supseteq \{x \in M(r_1) \mid x; M(r_2) \cap A(r) \neq \emptyset\} && \text{if } r = r_1; r_2 \\ A(r_1) &\supseteq \{x \in M(r_1) \mid M(r_1^*); x; M(r_1^*) \cap A(r) \neq \emptyset\} && \text{if } r = r_1^+ \text{ or } r = r_1^*. \end{aligned}$$

The definition maintains the invariant $A(r) \subseteq M(r)$. If a pair (e_1, e_2) is relevant to relation $r = r_1^{-1}$, then (e_2, e_1) will be relevant to r_1 . We do not have to intersect $A(r)^{-1}$ with $M(r)^{-1}$ because $A(r) \subseteq M(r)$ ensures $A(r)^{-1} \subseteq M(r)^{-1}$. We can avoid the intersection with the may pairs for the next case as well. There, $A(r) \subseteq M(r)$ holds by the invariant and $M(r) = M(r_1) \cap M(r_2)$ by definition (see below). For union and the other case of subtraction, the intersection with $M(r_1)$ is necessary. There are symmetric definitions for union and intersection for r_2 . For a relation r_1 that occurs in a relational composition $r = r_1; r_2$, the pairs (e_1, e_3) become relevant if they may be composed with a pair (e_3, e_2) in r_2 to obtain a pair (e_1, e_2) relevant to r . Note that for r_2 we again need the may pairs. The definition for r_2 is similar. The definition for the (reflexive and) transitive closure follows the ideas for relational composition.

The definition of the may sets follows the syntax of the CAT memory model bottom-up. With $\oplus \in \{\cup, \cap, ;\}$ and $\otimes \in \{+, *, -1\}$, we have:

$$M(r_1 \oplus r_2) \supseteq M(r_1) \oplus M(r_2) \quad M(r^\otimes) \supseteq M(r)^\otimes \quad M(r_1 \setminus r_2) \supseteq M(r_1).$$

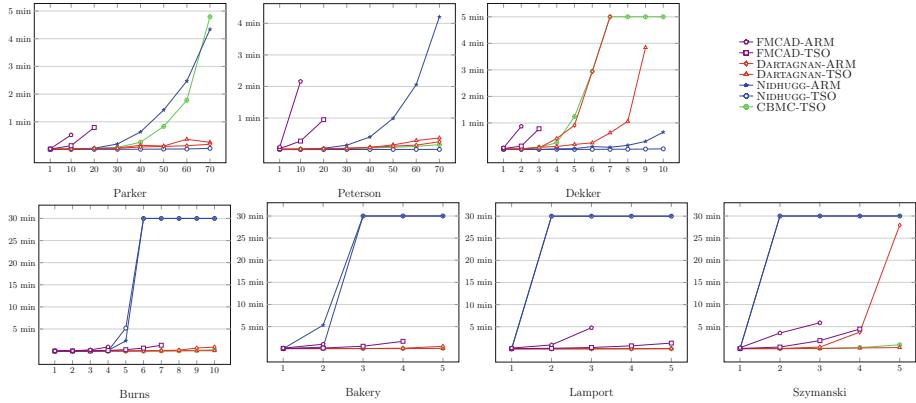


Fig. 1. Impact of the unrolling bound (x -axis) on the verification time (y -axis).

This simply executes the operator of the relation on the corresponding may sets. Subtraction ($r_1 \setminus r_2$) is the exception, it is not sound to over-approximate r_2 .

At the bottom level, the may sets are determined by the base relations. They depend on the shape of the relations and the positions of the events in the control flow. The relations `loc`, `co` and `rf` are concerned with memory accesses. What makes it difficult to approximate these relations is our support for pointers and pointer arithmetic. Without further information, we have to conservatively assume that a memory event may access any address. To improve the precision of the may sets for `loc`, `co`, and `rf`, our fixpoint computation incorporates a *may-alias analysis*. We use a control-flow insensitive Andersen-style analysis [17]. It incurs only a small overhead and produces a close over-approximation of the may sets. The analysis returns¹ a set of pairs of memory events $PTS \subseteq (\mathbb{W} \cup \mathbb{R}) \times (\mathbb{W} \cup \mathbb{R})$ such that every pair of events outside PTS definitely accesses different addresses. Here, \mathbb{W} are the store events in the program and \mathbb{R} are the loads. Note that the analysis has to be control-flow insensitive as the given memory model may be very weak [10]. We have $M(\text{loc}) \supseteq PTS$. Similarly, $M(\text{co})$ and $M(\text{rf})$ are defined by PTS restricted to $(\mathbb{W} \times \mathbb{W})$ and $(\mathbb{W} \times \mathbb{R})$, respectively.

We stress the importance of the alias analysis for our relation analysis: `loc`, `co`, and `rf` are frequently used as building blocks of composite relations. Excessive may sets will therefore negatively affect the over-approximations of virtually all relations in a memory model, and keep the overall encoding unnecessarily large.

Illustration. We illustrate the relation analysis on the example from the introduction. Consider constraint `acyclic ghb-tso`. The computation of the active set for the relation `ghb-tso` triggers the calculation of the may set, following the inequality $A(\text{ghb-tso}) \supseteq M(\text{ghb-tso}) \cap M(\text{ghb-tso}^+)^{-1}$. The may set is the union of the may sets for the subrelations, shown by colored (dotted and solid) edges.

¹ This is a simplification, Andersen returns points-to sets, and we check by an intersection $PTS(r_1) \cap PTS(r_2)$ whether two registers may alias.

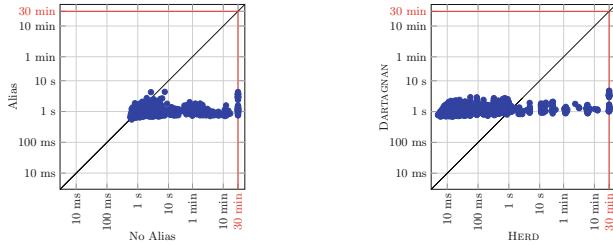


Fig. 2. Execution times (logarithmic scale) on LINUX kernel litmus tests: impact of alias analysis (left) and comparison against HERD (right).

The intersection yields the edges that may lie on cycles of ghb-tso. They are drawn in solid. These solid edges in $A(\text{ghb-tso})$ are propagated down to the sub-relations. For example, $A(\text{po-tso}) \supseteq A(\text{ghb-tso}) \cap M(\text{po-tso})$ yields the solid black edges.

4 Experiments

We compare DARTAGNAN to CBMC [13] and NIDHUGG [5, 9], both model-specific tools, and to HERD [4, 16] and the DARTAGNAN FMCAD-18 version [3, 28] (without relation analysis), both taking CAT models as inputs. We also evaluate the impact of the alias analysis on the execution time.

Benchmarks. For CBMC, NIDHUGG, and the FMCAD-18 DARTAGNAN, we evaluate the performance on 7 mutual exclusion benchmarks executed on TSO (all tools) and a subset of ARMv7 (only NIDHUGG and DARTAGNAN). The results on POWER are similar to those on ARM and thus omitted. We excluded HERD from this experiment since it did not scale even for small unrolling bounds [28]. We set a 5 min timeout for Parker, Dekker, and Peterson as this is sufficient to show the trends in the runtimes, and a 30 min timeout for the remaining benchmarks. To compare against HERD, and to evaluate the impact of the alias analysis, we run 4751 LINUX kernel litmus tests (all tests from [1] without LINUX spinlocks). The tests contain kernel primitives, such as RCU, on the LINUX kernel model. We set a 30 min timeout.

Evaluation. The times for CBMC, NIDHUGG-ARM, and the FMCAD-2018 version of DARTAGNAN grow exponentially for Parker (see Fig. 1). The growth in CBMC and FMCAD-2018 is due to the explosion of the encoding. For the latter, the solver runs out of memory with unrolling bounds 20 (TSO) and 10 (ARM). For NIDHUGG-ARM, the tool explores many unnecessary executions. The verification times for NIDHUGG-TSO and the current version of DARTAGNAN grow linearly. The latter is due to the relation analysis. For Peterson, the results are similar except for CBMC, which matches DARTAGNAN's performance.

For Dekker, NIDHUGG outperforms both CBMC and DARTAGNAN. This is because the number of executions grows slowly compared to the explosion of the

number of instructions. The executions in both memory models coincide, making the performance on ARM comparable to that on TSO for NIDHUGG. The difference is due to the optimal exploration in TSO, but not in ARM. Relation analysis has some impact on the performance (see FMCAD-2018 vs. DARTAGNAN), but the encoding size still grows faster than the number of executions.

The benchmarks Burns, Bakery, and Lamport demonstrate the opposite trend: the number of executions grows much faster than the size of the encoding. Here, CBMC and DARTAGNAN outperform NIDHUGG. Notice that for Burns, NIDHUGG performs better on ARM than on TSO with unrolling bound 5. This is counter-intuitive since one expects more executions on ARM. Although the number of executions coincide, the exploration time is higher on TSO due to a different search algorithm. For Szymanski, similar results hold except for DARTAGNAN-ARM where the encoding grows exponentially.

Figure 2 (left) shows the verification times for the current version of DARTAGNAN with and without alias analysis. The alias analysis results in a speed-up of more than two orders of magnitude in benchmarks with several threads accessing up to 18 locations. Figure 2 (right) compares the performance of DARTAGNAN against HERD. We used the Knaster-Tarski encoding and alias analysis since they yield the best performance. HERD outperforms DARTAGNAN on small test instances (less than 1 s execution time). This is due to the JVM startup time and the preprocessing costs of DARTAGNAN. However, on large benchmarks, HERD times out while DARTAGNAN takes less than 10 s.

References

1. Linux kernel litmus test suite. <https://github.com/paulmckrcu/litmus>
2. Linux Memory Model. <https://github.com/torvalds/linux/tree/master/tools/memory-model>
3. The Dat3M tool suite. <https://github.com/hernanponcedeleon/Dat3M>
4. The herdtools7 tool suite. <https://github.com/herd/herdtools7>
5. Abdulla, P.A., Aronis, S., Atig, M.F., Jonsson, B., Leonardsson, C., Sagonas, K.: Stateless model checking for TSO and PSO. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 353–367. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_28
6. Abdulla, P.A., Atig, M.F., Chen, Y.-F., Leonardsson, C., Rezine, A.: Automatic fence insertion in integer programs via predicate abstraction. In: Miné, A., Schmidt, D. (eds.) SAS 2012. LNCS, vol. 7460, pp. 164–180. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33125-1_13
7. Abdulla, P.A., Atig, M.F., Chen, Y.-F., Leonardsson, C., Rezine, A.: Counterexample guided fence insertion under TSO. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 204–219. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28756-5_15
8. Abdulla, P.A., Atig, M.F., Chen, Y.-F., Leonardsson, C., Rezine, A.: MEMORAX, a precise and sound tool for automatic fence insertion under TSO. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 530–536. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_37

9. Abdulla, P.A., Atig, M.F., Jonsson, B., Leonardsson, C.: Stateless model checking for POWER. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 134–156. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_8
10. Alglave, J., Kroening, D., Lugton, J., Nimal, V., Tautschnig, M.: Soundness of data flow analyses for weak memory models. In: Yang, H. (ed.) APLAS 2011. LNCS, vol. 7078, pp. 272–288. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25318-8_21
11. Alglave, Jade: A Shared Memory Poetics. Thèse de doctorat, L'université Paris Denis Diderot (2010)
12. Alglave, J., Cousot, P., Maranget, L.: Syntax and semantics of the weak consistency model specification language CAT. CoRR, [arXiv:1608.07531](https://arxiv.org/abs/1608.07531) (2016)
13. Alglave, J., Kroening, D., Tautschnig, M.: Partial orders for efficient bounded model checking of concurrent software. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 141–157. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_9
14. Alglave, J., Maranget, L.: Stability in weak memory models. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 50–66. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_6
15. Alglave, J., Maranget, L., McKenney, P.E., Parri, A., Stern, A.S.: Frightening small children and disconcerting grown-ups: Concurrency in the linux kernel. In: ASPLOS, pp. 405–418. ACM (2018)
16. Alglave, J., Maranget, L., Tautschnig, M.: Herding cats: Modelling, simulation, testing, and data mining for weak memory. ACM Trans. Program. Lang. Syst. **36**(2), 7:1–7:74 (2014)
17. Andersen, L.O.: Program Analysis and Specialization for the C Programming Language. PhD thesis, University of Copenhagen (1994)
18. Batty, M., Donaldson, A.F., Wickerson, J.: Overhauling SC atomics in C11 and OpenCL. In: POPL, pp. 634–648. ACM (2016)
19. Bouajjani, A., Derevenetc, E., Meyer, R.: Checking and enforcing robustness against TSO. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 533–553. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_29
20. Chong, N., Sorensen, T., Wickerson, J.: The semantics of transactions and weak memory in x86, Power, ARM, and C++. In: PLDI, pp. 211–225. ACM (2018)
21. Clarke, E.M., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Form. Methods Syst. Des. **19**(1), 7–34 (2001)
22. Kokologiannakis, M., Lahav, O., Sagonas, K., Vafeiadis, V.: Effective stateless model checking for C/C++ concurrency. PACMPL **2**(POPL), 17:1–7:32 (2018)
23. Kuperstein, M., Vechev, M.T., Yahav, E.: Automatic inference of memory fences. SIGACT News **43**(2), 108–123 (2012)
24. Lahav, O., Vafeiadis, V., Kang, J., Hur, C.-H., Kil, Dreyer, D.: Repairing sequential consistency in C/C++11. In: PLDI, pp. 618–632. ACM (2017)
25. Liu, F., Nedev, N., Prasadnikov, N., Vechev, M.T., Yahav, E.: Dynamic synthesis for relaxed memory models. In: PLDI, pp. 429–440. ACM (2012)
26. McKenney, P.E., Slingwine, J.: Read-copy update: Using execution history to solve concurrency problems. In: Parallel and Distributed Computing and Systems, pp 509–518 (1998)
27. Ponce-de-León, H., Furbach, F., Heljanko, K., Meyer, R.: Portability analysis for weak memory models PORTHOS: **One Tool for all Models**. In: Ranzato, F. (ed.) SAS 2017. LNCS, vol. 10422, pp. 299–320. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66706-5_15

28. Ponce de León, H., Furbach, F., Heljanko, K., Meyer, R.: BMC with memory models as modules. In: FMCAD, pp. 1–9. IEEE (2018)
29. Pulte, C., Flur, S., Deacon, W., French, J., Sarkar, S., Sewell, P.: Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. PACMPL **2**(POPL), 19:1–19:29 (2018)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





When Human Intuition Fails: Using Formal Methods to Find an Error in the “Proof” of a Multi-agent Protocol

Jennifer A. Davis¹, Laura R. Humphrey^{2(✉)}, and Derek B. Kingston³

¹ Collins Aerospace, Cedar Rapids, IA 52498, USA
jen.davis@collins.com

² Air Force Research Lab, Dayton, OH 45433, USA
laura.humphrey@us.af.mil

³ Aurora Flight Sciences, Manassas, VA 20110, USA
kingston.derek@aurora.aero

Abstract. Designing protocols for multi-agent interaction that achieve the desired behavior is a challenging and error-prone process. The standard practice is to manually develop proofs of protocol correctness that rely on human intuition and require significant effort to develop. Even then, proofs can have mistakes that may go unnoticed after peer review, modeling and simulation, and testing. The use of formal methods can reduce the potential for such errors. In this paper, we discuss our experience applying model checking to a previously published multi-agent protocol for unmanned air vehicles. The original publication provides a compelling proof of correctness, along with extensive simulation results to support it. However, analysis through model checking found an error in one of the proof’s main lemmas. In this paper, we start by providing an overview of the protocol and its original “proof” of correctness, which represents the standard practice in multi-agent protocol design. We then describe how we modeled the protocol for a three-vehicle system in a model checker, the counterexample it returned, and the insight this counterexample provided. We also discuss benefits, limitations, and lessons learned from this exercise, as well as what future efforts would be needed to fully verify the protocol for an arbitrary number of vehicles.

Keywords: Multi-agent systems · Distributed systems · Autonomy · Model checking

1 Introduction

Many robotics applications require multi-agent interaction. However, designing protocols for multi-agent interaction that achieve the desired behavior can be

D. B. Kingston—Supported by AFRL/RQ contract #FA8650-17-F-2220 and AFOSR award #17RQCOR417. DISTRIBUTION A. Approved for public release: distribution unlimited. Case #88ABW-2018-4275.

This is a U.S. government work and not under copyright protection in the U.S.; foreign copyright protection may apply 2019

I. Dillig and S. Tasiran (Eds.): CAV 2019, LNCS 11561, pp. 366–375, 2019.

https://doi.org/10.1007/978-3-030-25540-4_20

challenging. The design process is often manual, i.e. performed by humans, and generally involves creating mathematical models of possible agent behaviors and candidate protocols, then manually developing a proof that the candidate protocols are correct with respect to the desired behavior. However, human-generated proofs can have mistakes that may go unnoticed even after peer review, modeling and simulation, and testing of the resulting system.

Formal methods have the potential to reduce such errors. However, while the use of formal methods in multi-agent system design is increasing [2, 6, 8, 11], it is our experience that manual approaches are still the norm. Here, we hope to motivate the use of formal methods for multi-agent system design by demonstrating their value in a case study involving a manually designed decentralized protocol for dividing surveillance of a perimeter across multiple unmanned aerial vehicles (UAVs). This protocol, called the Decentralized Perimeter Surveillance System (DPSS), was previously published in 2008 [10], has received close to 200 citations to date, and provides a compelling “proof” of correctness backed by extensive simulation results.

We start in Sect. 2 by giving an overview of DPSS, the convergence bounds that comprise part of its specification, and the original “proof” of correctness. In Sect. 3, we give an overview of the three-UAV DPSS model we developed in the Assume Guarantee REasoning Environment (AGREE) model checker [3]. In Sect. 4, we present the analysis results returned by AGREE, including a counterexample to one of the convergence bounds. Section 5 concludes with a discussion of benefits, challenges, and limitations of our modeling process and how to help overcome them, and what future work would be required to modify and fully verify DPSS for an arbitrary number of UAVs.

2 Decentralized Perimeter Surveillance System (DPSS)

UAVs can be used to perform continual, repeated surveillance of a large perimeter. In such cases, more frequent coverage of points along the perimeter can be achieved by evenly dividing surveillance of it across multiple UAVs. However, coordinating this division is challenging in practice for several reasons. First, the exact location and length of the perimeter may not be known *a priori*, and it may change over time, as in a growing forest fire or oil spill. Second, UAVs might go offline and come back online, e.g. for refueling or repairs. Third, inter-UAV communication is unreliable, so it is not always possible to immediately communicate local information about perimeter or UAV changes. However, such information is needed to maintain an even division of the perimeter as changes occur. DPSS provides a method to solve this problem with minimal inter-UAV communication for perimeters that are isomorphic to a line segment.

Let the perimeter start as a line segment along the x -axis with its left endpoint at $x = 0$ and its right at $x = P$. Let N be the number of UAVs in the system or on the “team,” indexed from left to right as $1, \dots, N$. Divide the perimeter into segments of length P/N , one per UAV. Then the optimal configuration of DPSS as depicted in Fig. 1 is defined as follows (see Ref. [10] for discussion of why this definition is desirable).

Definition 1. Consider two sets of perimeter locations: (1) $\lfloor i + \frac{1}{2}(-1)^i \rfloor P/N$ and (2) $\lfloor i - \frac{1}{2}(-1)^i \rfloor P/N$, where $\lfloor \cdot \rfloor$ returns the largest integer less than or equal to its argument. The optimal configuration is realized when UAVs synchronously oscillate between these two sets of locations, each moving at constant speed V .

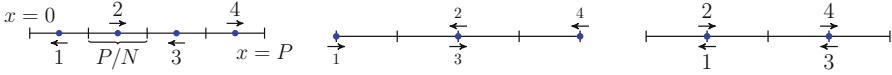


Fig. 1. Optimal DPSS configuration, in which UAVs are evenly spaced along the perimeter and synchronously oscillate between segment boundaries.

The goal of DPSS is to achieve the optimal configuration in the steady state, i.e. when the perimeter and involved UAVs remain constant. The DPSS protocol itself is relatively simple. Each UAV i stores a vector $\xi_i = [P_{R_i} \ P_{L_i} \ N_{R_i} \ N_{L_i}]^T$ of coordination variables that capture its beliefs (which may be incorrect) about perimeter length P_{R_i} and P_{L_i} and number of UAVs N_{R_i} and N_{L_i} to its right and left. When neighboring UAVs meet, “left” UAV i learns updated values for its “right” variables $P'_{R_i} = P_{R_{i+1}}$ and $N'_{R_i} = N_{R_{i+1}} + 1$ from “right” UAV $i + 1$, and likewise UAV $i + 1$ updates its “left” variables $P'_{L_{i+1}} = P_{L_i}$ and $N'_{L_{i+1}} = N_{L_i} + 1$. While values for these variables may still be incorrect, the two UAVs will at least have matching coordination variables and thus a consistent estimate of their shared segment boundary. The two UAVs then “escort” each other to their estimated shared segment boundary, then split apart to surveil their own segment. Note that UAVs only change direction when they reach a perimeter endpoint or when starting or stopping an escort, which means a UAV will travel outside its segment unless another UAV arrives at the segment boundary at the same time (or the end of the segment is a perimeter endpoint).

Eventually, leftmost UAV 1 will discover the actual left perimeter endpoint, accurately set $N_{L_1} = 0$ and $P_{L_1} = 0$, then turn around and update P_{L_1} continuously as it moves. A similar situation holds for rightmost UAV n . Accurate information will be passed along to other UAVs as they meet, and eventually all UAVs will have correct coordination variables and segment boundary estimates. Since UAVs also escort each other to shared segment boundaries whenever they meet, eventually the system reaches the optimal configuration, in which UAVs oscillate between their true shared segment boundaries.

An important question is how long it takes DPSS to converge to the optimal configuration. Each time the perimeter or number of UAVs changes, it is as if the system is reinitialized; UAVs no longer have correct coordination variables and so the system is no longer converged. However, if DPSS is able to re-converge relatively quickly, it will often be in its converged state.

Ref. [10] claims that DPSS converges within $5T$, where $T = P/V$ is the time it would take a single UAV to traverse the entire perimeter if there were no other UAVs in the system. It describes DPSS as two algorithms: Algorithm A, in

which UAVs start with correct coordination variables, and Algorithm B, in which they do not. The proof strategy is then to argue that Algorithm A converges in $2T$ (Theorem 1) and Algorithm B achieves correct coordination variables in $3T$ (Lemma 1)¹. At that point, Algorithm B converts to Algorithm A, so the total convergence time is $2T + 3T = 5T$ (Theorem 2)².

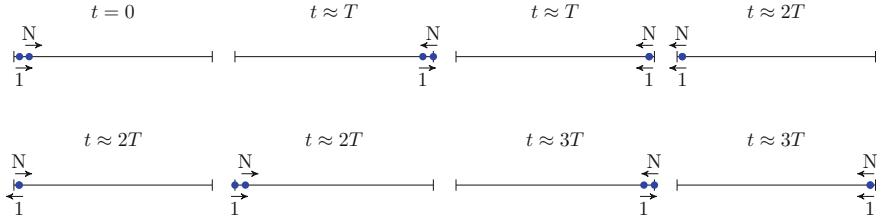


Fig. 2. Claimed worst-case coordination variable convergence for Algorithm B.

Informally, the original argument for Lemma 1 is that information takes time T to travel along the perimeter. The worst case occurs when all UAVs start near one end of the perimeter, e.g. the left endpoint, so that the rightmost UAV N reaches the right endpoint around time T . UAV N then turns around and through a fast series of meetings, correct “right” coordination variables are propagated to the other UAVs, all of which then start moving left. Due to incorrect “left” coordination variables, UAV $N - 1$ and UAV N might think their shared segment boundary is infinitesimally close to the left endpoint. The UAVs travel left until they are almost at the left perimeter endpoint around time $2T$. However, since UAV N thinks its segment boundary is near the left endpoint, it ends its escort and goes right without learning the true location of the left perimeter endpoint. Leftmost UAV 1 learns the true location of the left perimeter endpoint and this information will be passed to the other UAVs as they meet, but the information will have to travel the perimeter once again to reach the rightmost UAV N around time $3T$. This situation is depicted in Fig. 2.

Through model checking, we were able to find a counterexample to this claimed bound, which will be presented in Sect. 4. But first, we overview the model used for analysis through model checking.

3 Formal Models

We briefly overview the formal models developed in AGREE for a three-UAV version of DPSS as described by Algorithm B. Models for Algorithm A and

¹ We label this Lemma 1 for convenience; it is unlabeled in [10].

² A version of the original proof is on GitHub [1] in file dpssOriginalProof.pdf.

Algorithm B along with a more detailed description of the Algorithm B model are available on GitHub [1]³.

AGREE is an infinite-state model checker capable of analyzing systems with real-valued variables, as is the case with DPSS. AGREE uses assume/guarantee reasoning to verify properties of architectures modeled as a top-level system with multiple lower-level components, each having a formally specified assume/guarantee contract. Each contract consists of a set of assumptions on the inputs and guarantees on the outputs, where inputs and outputs can be reals, integers, or booleans. System assumptions and component assume/guarantee contracts are assumed to be true. AGREE then attempts to verify that (a) component assumptions hold given system assumptions, and (b) system guarantees hold given component guarantees. AGREE poses this verification problem as a satisfiability modulo theory (SMT) problem [4] and uses a k-induction model checking approach [7] to search for counterexamples that violate system-level guarantees given system-level assumptions and component-level assume/guarantee contracts. The language used by AGREE is an “annex” to the Architecture Analysis and Design Language (AADL) [5].

AGREE’s ability to analyze systems modeled as a top-level system with multiple lower-level components provides a natural fit for DPSS. The three-UAV AGREE DPSS model consists of a single top-level system model, which we call the “System,” and a component-level UAV model that is instantiated three times, which we call the “UAV(s).” The System essentially coordinates a discrete event simulation of the UAVs as they execute the DPSS protocol, where events include a UAV reaching a perimeter endpoint or two UAVs starting or stopping an escort. In the initial state, the System sets valid ranges for each UAV’s initial position through assumptions that constrain the UAVs to be initialized between the perimeter endpoints and ordered by ID number from left to right. System assumptions also constrain UAV initial directions to be either left or right (though a UAV might have to immediately change this value, e.g., if it is initialized at the left endpoint headed left). These values become inputs to the UAVs. The System determines values for other UAV inputs, including whether a UAV is co-located with its right or left neighbor and the true values for the left and right perimeter endpoints. Note the true perimeter endpoints are only used by the UAVs to check whether they have reached the end of the perimeter, not to calculate boundary segment endpoints. The System also establishes data ports between UAVs, so that each UAV can receive updated coordination variable values from its left or right neighbor as inputs and use them (but only if they are co-located).

The last System output that serves as a UAV input is the position of the UAV. At initialization and after each event, the System uses the globally known constant UAV speed V and other information from each UAV to determine the amount of time δt until the next event, and then it updates the position of each

³ AADL projects are in AADL_sandbox_projects. Algorithm A and B models for three UAVs are in projects DPSS-3-AlgA-for-paper and DPSS-3-AlgB-for-paper. A description of the Algorithm B model is in file modelAlgorithmB.pdf.

UAV. Determining the time of the next event requires knowing the direction and next anticipated “goal” location of each UAV, e.g. estimated perimeter endpoint or shared segment boundary. Each UAV outputs these values, which become inputs to the System. Each UAV also outputs its coordination variables P_{R_i} , P_{L_i} , N_{R_i} , and N_{L_i} , which become System inputs that are used in System guarantees that formalize Theorem 1, Lemma 1, and Theorem 2 of Sect. 2. Note that we bound integers N_{R_i} and N_{L_i} because in order to calculate estimated boundary segments, which requires dividing perimeter length by the number of UAVs, we must implement a lookup table that copies the values of N_{R_i} and N_{L_i} to real-valued versions of these variables. This is due to an interaction between AGREE and the Z3 SMT solver [4] used by AGREE. If we directly cast N_{R_i} and N_{L_i} to real values in AGREE, they are encoded in Z3 using the `to_real` function. Perimeter values P_{R_i} and P_{L_i} are directly declared as reals. However, Z3 views integers converted by the `to_real` function as constrained to have integer values, so it cannot use the specialized solver for reals that is able to analyze this model.

4 Formal Analysis Results

In this section, we discuss the analysis results provided by AGREE for Algorithm A and Algorithm B, though we focus on Algorithm B.

Algorithm A: Using AGREE configured to utilize the JKInd k-induction model checker [7] and the Z3 SMT solver, we have proven Theorem 1, that Algorithm A converges within $2T$, for $N = 1, 2, 3, 4, 5$, and 6 UAVs. Computation time prevented us from analyzing more than six UAVs. For reference, $N = 1$ through $N = 4$ ran in under 10 min each on a laptop with two cores and 8 GB RAM. The same laptop analyzed $N = 5$ overnight. For $N = 6$, the analysis took approximately twenty days on a computer with 40 cores and 128 GB memory.

Algorithm B: We were able to prove Theorem 2, that DPSS converges within $5T$, for $N = 1, 2$, and 3 UAVs and with each UAV’s coordination variables N_{R_i} and N_{L_i} bounded between 0 and 20. In fact, we found the convergence time to be within $(4 + \frac{1}{3}T)$. However, AGREE produced a counterexample to Lemma 1, that every UAV obtains correct coordination variables within $3T$, for $N = 3$. In fact, we incrementally increased this bound and found counterexamples up to $(3 + \frac{1}{2}T)$ but that convergence is guaranteed in $(3 + \frac{2}{3}T)$.

One of the shorter counterexamples provided by AGREE shows the UAVs obtaining correct coordination variables in $3.0129T$. Full details are available on GitHub [1]⁴, but we outline the steps in Fig. 3. In this counterexample, UAV 1 starts very close to the left perimeter heading right, and UAVs 2 and 3 start in the middle of segment 3 headed left. UAVs 1 and 2 meet near the middle of the perimeter and head left toward what they believe to be their shared segment boundary. This is very close to the left perimeter endpoint because, due to initial conditions, they believe the left perimeter endpoint to be much

⁴ A spreadsheet with counterexample values for all model variables is located under `AADL-sandbox-projects/DPSS-3-AlgB-for-paper/results_20180815_eispi`.

farther away than it actually is. Then they split, and UAV 1 learns where the left perimeter endpoint actually is, but UAV 2 does not. UAV 2 heads right and meets UAV 3 shortly afterward, and they move to what they believe to be their shared segment boundary, which is likewise very close to the right perimeter endpoint. Then they split, and UAV 3 learns where the right perimeter endpoint is, but UAV 2 does not. UAV 2 heads left, meets UAV 1 shortly after, and learns correct “left” coordination variables. However, UAV 2 still believes the right perimeter endpoint to be farther away than it actually is, so UAV 1 and 2 estimate their shared segment boundary to be near the middle of the perimeter. They then head toward this point and split apart, with UAV 1 headed left and still not having correct “right” coordination variables. UAV 2 and 3 then meet, exchange information, and now both have correct coordination variables. They go to their actual shared boundary, split apart, and UAV 2 heads left toward UAV 1. UAV 1 and 2 then meet on segment 1, exchange information, and now all UAVs have correct coordination variables.

The counterexample reveals a key intuition that was missing in Lemma 1. The original argument did not fully consider the effects of initial conditions and so only considered a case in which UAVs came close to *one* end of the perimeter without actually reaching it. The counterexample shows it can happen at *both* ends if initial conditions cause the UAVs to believe the perimeter endpoints to be farther away than they actually are. This could happen if the perimeter were to quickly shrink, causing the system to essentially “reinitialize” with incorrect coordination variables.

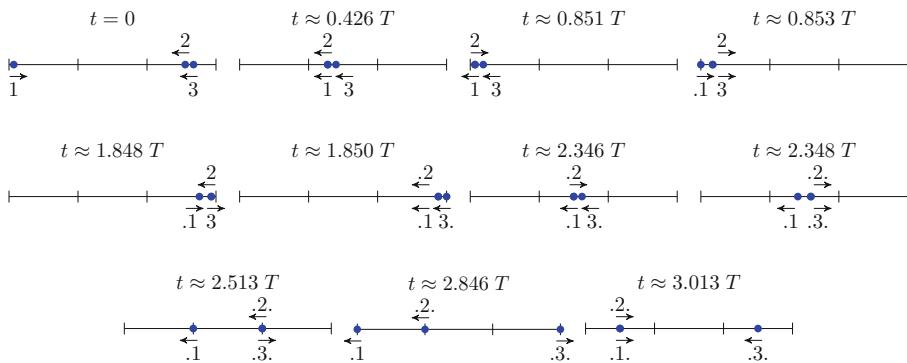


Fig. 3. Counterexample to Lemma 1. Dots to the left of a UAV number indicate it has correct “left” variables, and likewise for the right.

Analysis for three UAVs for Algorithm B completed in 18 days on a machine with 256 GB RAM and 80 cores.

5 Discussion and Conclusions

Formal modeling and analysis through AGREE had many benefits. First, it allowed us to analyze DPSS, a decentralized protocol for distributing a surveillance task across multiple UAVs. Though the original publication on DPSS provided a convincing human-generated proof and simulation results to support claims about its convergence bounds, analysis revealed that one of the key lemmas was incorrect. Furthermore, the counterexample returned by AGREE provided insight into why it was incorrect. Second, formal modeling in and of itself allowed us to find what were essentially technical typos in the original paper. For example, the formula for dividing the perimeter across UAVs only accounted for changes in estimates of the right perimeter endpoint and not the left, so we corrected the formula for our model. We also discovered that certain key aspects of the protocol were underspecified. In particular, it is unclear what should happen if more than two UAVs meet at the same time. Analysis showed this occurring for as little as three UAVs in Algorithm B, and simulations in the original paper showed this happening frequently, but this behavior was not explicitly described. Here, we decided that if all three UAVs meet to the left of UAV 3’s estimated segment, UAV 3 immediately heads right and the other two follow the normal protocol to escort each other to their shared border. Otherwise, the UAVs all travel left together to the boundary between segments 2 and 3, then UAV 3 breaks off and heads right while the other two follow the normal protocol.

This brings us to a discussion of challenges and limitations. First, in terms of more than two UAVs meeting at a time, simulations in the original paper implement a more complex behavior in which UAVs head to the closest shared boundary and then split apart into smaller and smaller groups until reaching the standard case of two co-located UAVs. This behavior requires a more complex AGREE model that can track “cliques” of more than two UAVs, and it is difficult to validate the model due to long analysis run times. Second, we noted in Sect. 4 that in our model, UAV coordination variables N_{R_i} and N_{L_i} have an upper bound of 20. In fact, with an earlier upper bound of 3, we found the bound for Lemma 1 to be $(3 + \frac{1}{3})T$ and did not consider that it would depend on upper bounds for N_{R_i} and N_{L_i} . We therefore cannot conclude that even $(3 + \frac{2}{3})T$ is the convergence time for Lemma 1. Third and related to the last point, model checking with AGREE can only handle up to three UAVs for Algorithm B. Due to these limitations, we cannot say for sure what the upper bound for DPSS actually is, even if we believe it to be $5T$. If it is higher, then it takes DPSS longer to converge, meaning it can handle less frequent changes than originally believed. We are therefore attempting to transition to theorem provers such as ACL2 [9] and PVS [12] to develop a proof of convergence bounds for an arbitrary number of UAVs, upper bound on N_{R_i} and N_{L_i} , and perimeter length (which was set to a fixed size to make the model small enough to analyze).

In terms of recommendations and lessons learned, it was immensely useful to work with the author of DPSS to formalize our model. Multi-agent protocols like DPSS are inherently complex, and it is not surprising that the original paper contained some typos, underspecifications, and errors. In fact, the original paper

explains DPSS quite well and is mostly correct, but it is still challenging for formal methods experts to understand complex systems from other disciplines, so access to subject matter experts can greatly speed up formalization.

Acknowledgment. We thank John Backes for his guidance on efficiently modeling DPSS in AGREE and Aaron Fifarek for running some of the longer AGREE analyses.

References

1. OpenUxAS GitHub repository, dpssModel branch. <https://github.com/afrl-rq/OpenUxAS/tree/dpssModel>
2. Alur, R., Moarref, S., Topcu, U.: Compositional synthesis of reactive controllers for multi-agent systems. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 251–269. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_14
3. Cofer, D., Gacek, A., Miller, S., Whalen, M.W., LaValley, B., Sha, L.: Compositional verification of architectural models. In: Goodloe, A.E., Person, S. (eds.) NFM 2012. LNCS, vol. 7226, pp. 126–140. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28891-3_13
4. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
5. Feiler, P.H., Lewis, B.A., Vestal, S.: The SAE architecture analysis & design language (AADL): a standard for engineering performance critical systems. In: IEEE International Conference Computer Aided Control System Design, pp. 1206–1211. IEEE (2006)
6. Fisher, M., Dennis, L., Webster, M.: Verifying autonomous systems. Commun. ACM **56**(9), 84–93 (2013)
7. Gacek, A., Backes, J., Whalen, M., Wagner, L., Ghassabani, E.: The JKIND model checker. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10982, pp. 20–27. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96142-2_3
8. Guo, M., Tumova, J., Dimarogonas, D.V.: Cooperative decentralized multi-agent control under local LTL tasks and connectivity constraints. In: 2014 IEEE 53rd Annual Conference on Decision and Control (CDC), pp. 75–80. IEEE (2014)
9. Kaufmann, M., Moore, J.S.: An industrial strength theorem prover for a logic based on common lisp. IEEE Trans. Softw. Eng. **23**(4), 203–213 (1997)
10. Kingston, D., Beard, R.W., Holt, R.S.: Decentralized perimeter surveillance using a team of UAVs. IEEE Trans. Robot. **24**(6), 1394–1404 (2008)
11. Kupermann, O., Vardi, M.: Synthesizing distributed systems. In: Proceedings 16th Annual IEEE Symposium Logic in Computer Science, pp. 389–398. IEEE (2001)
12. Owre, S., Rushby, J.M., Shankar, N.: PVS: a prototype verification system. In: Kapur, D. (ed.) CADE 1992. LNCS, vol. 607, pp. 748–752. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-55602-8_217

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter’s Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter’s Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Extending NUXMV with Timed Transition Systems and Timed Temporal Properties

Alessandro Cimatti, Alberto Griggio,

Enrico Magnago, Marco Roveri^(✉),
and Stefano Tonetta



Fondazione Bruno Kessler, Trento, Italy
roveri@fbk.eu

Abstract. NUXMV is a well-known symbolic model checker, which implements various state-of-the-art algorithms for the analysis of finite- and infinite-state transition systems and temporal logics. In this paper, we present a new version that supports timed systems and logics over continuous super-dense semantics. The system specification was extended with clocks to constrain the timed evolution. The support for temporal properties has been expanded to include $\text{MTL}_{0,\infty}$ formulas with parametric intervals. The analysis is performed via a reduction to verification problems in the discrete-time case. The internal representation of traces has been extended to go beyond the lasso-shaped form, to take into account the possible divergence of clocks. We evaluated the new features by comparing NUXMV with other verification tools for timed automata and $\text{MTL}_{0,\infty}$, considering different benchmarks from the literature. The results show that NUXMV is competitive with and in many cases performs better than state-of-the-art tools, especially on validity problems for $\text{MTL}_{0,\infty}$.

1 Introduction

NUXMV [1] is a symbolic model checker for the analysis of synchronous finite- and infinite-state transition systems. For the finite-state case, NUXMV features strong verification engines based on state-of-the-art SAT-based algorithms. For the infinite-state case, NUXMV features SMT-based verification techniques, implemented through a tight integration with the MATHSAT5 solver [2]. NUXMV has taken part to recent editions of the hardware model checking competition, where it has shown to be very competitive with the state-of-the-art. NUXMV also compares well with other model checkers for infinite-state systems. Moreover, it has been successfully used in several application domains both in research and industrial settings. It is currently the core verification engine for many other tools (also industrial ones) for requirements analysis, contract based design, model checking of hybrid systems, safety assessment, and software model checking.

In this paper, we put emphasis on the novel extensions to NUXMV to support timed synchronous transition systems, which extend symbolically-represented infinite-state transition systems with clocks. The main novelties of this new version are the following. The NUXMV input language was extended to enable the description of symbolic

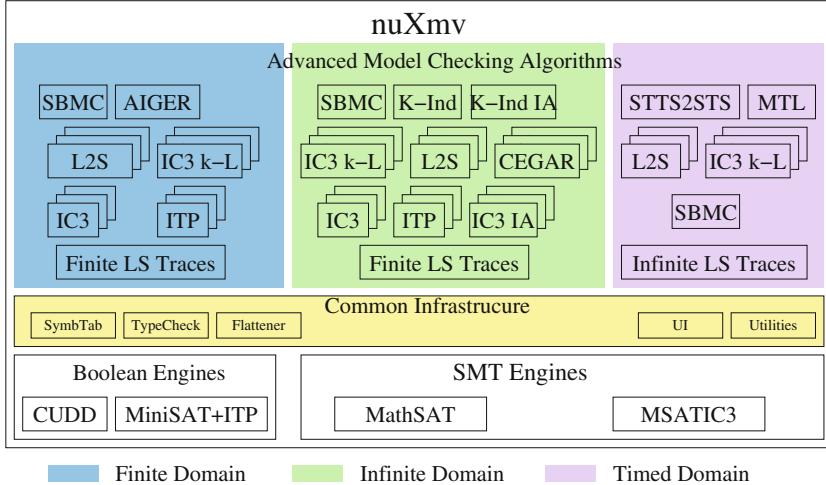


Fig. 1. The high level architecture of **NUXMV**.

synchronous timed transition systems with super-dense time semantics (where signals can have a sequence of values at any real time t). The support for temporal properties has been expanded to include $\text{MTL}_{0,\infty}$ formulas with parametric intervals [3,4]. Therefore, **NUXMV** now supports model checking of invariant, LTL and $\text{MTL}_{0,\infty}$ properties over (symbolic) timed transition systems, as well as validity/satisfiability checking of LTL and $\text{MTL}_{0,\infty}$ formulas. This is done via a correct and complete reduction to verification problems in the discrete-time case (thus allowing for the use of mature and efficient verification engines). In order to represent and find infinite traces where clocks may diverge, we extended the representation for lasso-shape traces (over discrete semantics) and we modified the bounded model checking algorithm to properly encode timed traces. We remark that, **NUXMV** is more expressive than timed automata, since the native management of time is added on top of an infinite state transition system. This makes it straightforward to encode stopwatches and comparison between clocks. We carried out an experimental evaluation comparing **NUXMV** with other state-of-the-art verification tools for timed automata, considering different benchmarks taken from competitor tools distributions.

2 Software Architecture

The high level architecture of **NUXMV** is depicted in Fig. 1. For symbolic transition systems **NUXMV** behaves like the previous version of the system [1], thus allowing for full backward compatibility (apart from some new reserved keywords). It provides the user with all the basic model checking algorithms for finite domains both using BDDs (using CUDD [5]) and SAT (e.g. MINISAT [6]). It supports various SMT-based model checking algorithms (implemented through a tight integration with the MATHSAT5 solver [2]) for the analysis of finite and infinite state systems (e.g. IC3 [7–9], k-liveness [10],

liveness to safety [11]). We refer the reader to [1] for a thorough discussion of these consolidated functionalities for the discrete-time setting.

```

1 @TIME.DOMAIN continuous — annotation to specify the time semantics, in this case dense time
2
3 MODULE main
4 FROZENVAR p: real; INIT p > 0 — parameter
5 VAR i: real; — input of the sensor
6 VAR s: Sensor(i);
7 VAR m: Monitor(s,o,p);
8
9 LTLSPEC G ( s.fault -> F [0,p] m.alarm ) — any fault is detected in p timed units
10
11 MODULE Sensor(i)
12 VAR o: real;
13 VAR fault: boolean;
14 TRANS !fault -> next(o) = i — if not faulty, the sensor provides in output directly the input
15 TRANS fault -> next(o) = o — if faulty, the sensor output is stuck at the last value
16 TRANS fault -> next(fault) — the fault is permanent
17
18 MODULE Monitor (i,p)
19 VAR previous.value: real;
20 VAR c: clock;
21 VAR alarm: boolean;
22 INIT c=0 & previous.value = i & !alarm
23 INVAR TRUE -> c <= p
24 TRANS time <= p | time >= p
25 TRANS (c = p & next(c) = 0 & next(previous.value) = i) | — the monitor reads the sensor every p time units
26 (c <= p & next(c) = c & next(previous.value) = previous.value)
27 TRANS next(alarm) <-> (alarm | i=previous.value) — alarm raised when the same value read twice consecutively

```

Fig. 2. A simple TIMED-NUXMV program.

To support the specification and model checking of invariant, LTL and $MTL_{0,\infty}$ properties for timed transitions systems, and for the validity checking of properties over dense time semantics, NUXMV has been extended w.r.t. [1] as discussed here after.

- We extended the parser to allow the user to choose the time semantics to use for the read model. Depending on the time model some parse constructs and checks are enabled and/or disabled. For instance, variables of type clock and $MTL_{0,\infty}$ properties are only allowed if the dense time semantics has been specified. By default the system uses the discrete time semantics of the original NUXMV. Notice also that, depending on the specified semantics, the commands available to the user change to allow only the analyses supported for the chosen semantics.
- We extended the parser to support the specification of symbolic timed automata (definition of clock variables, specification of urgent transitions and state invariants, etc.). Moreover, we extended the parser to allow for the specification of $MTL_{0,\infty}$ properties, and we extended the LTL bounded operators not only to contain constants, but also complex expressions over clock variables. See Fig. 2 for a simple example showing some of the new language constructs.
- We extended the symbol table to support the specification of clock variables, and we extended the type checker to properly handle the new defined variables, expression types and language constructs.
- We added new modules for the encoding of the symbolic timed automata into equivalent transition systems to verify with the existing algorithms of NUXMV.
- We extended the traces for NUXMV to support timed traces (lasso-shaped traces where some clock variables may diverge).

- We modified the encoding for the loops in the bounded model checking algorithms to take into account that traces may contain diverging variables to allow for the verification and validation of LTL and MTL_{0,∞} properties.

For portability, NUXMV has been developed mainly in standard C with some new parts in standard C++. It compiles and executes on Linux, MS Windows, and MacOS.

3 Language Extensions

Timed Transition Systems. Discrete-time transition systems are described in NUXMV by a set V of variables, an initial condition $I(V)$, a transition condition $T(V, V')$ and an invariant condition $Z(V)$. Variables are introduced with the keyword `VAR` and can have type Boolean, scalar, integer, real or array. The initial and the invariant conditions are introduced with the keyword `INIT` and `INVAR` and are expressions over the variables in V . The transition condition is introduced with `TRANS` and is an expression over variables in V and V' , where for each variable v in V , V' contains the “next” version denoted in the language by $\text{next}(v)$. Expressions may use standard symbols in the theory associated to the variable types and user-defined rigid functions that are declared with the keyword `FUN`.

The input language of NUXMV has been extended to allow the specification of timed transition systems (TTS), which are enabled by the annotation `@TIME_DOMAIN continuous` at the beginning of a model description.

Besides the standard types, in the timed case, state variables can be declared of type `clock`. All variables of type different from `clock` are discrete variables.

The language provides a built-in `clock` variable, accessible through the reserved keyword `time`. It represents the amount of time elapsed from the initial state until now. `time` is initialized to 0 and its value does not change in discrete transitions. While all other `clock` variables can be used in any expression in the model definition, `time` can be used only in comparison with constants.

Initial, transition, and invariant conditions are specified in NUXMV with the keywords `INIT`, `TRANS`, and `INVAR`, as in the discrete case. In particular, `TRANS` allows to specify “arbitrary” clock resets. Like all other NUXMV state variables, if a clock is not constrained during a discrete transition, its next value is chosen non-deterministically.

Clock variables can be used in `INVAR` only in the form $\varphi \rightarrow \phi$, where φ is a formula built using only the discrete variables and ϕ is convex over the clock variables. This closely maps the concept of location invariant described for timed automata: all locations satisfying φ have invariant ϕ .

An additional constraint, not allowed in the discrete-time case, is introduced with the keyword `URGENT` followed by a predicate over the discrete variables, which allows to specify a set of locations in which time cannot elapse.

Comparison with Timed Automata. Timed automata can be represented by TTSs by simply introducing a variable representing the locations of the automaton. Note that, in TTS, it is possible to express any kind of constraint over clock variables in discrete transitions, while in timed automata it is only possible to reset them to 0 in transitions or compare them to constants in guards. Moreover, the discrete variables of a

timed automaton always have finite domain, while in TTSs, also the discrete variables might have an infinite domain. This additional expressiveness allows to describe more complex behaviors (e.g. it is straightforward to encode stopwatches and comparison between clocks) losing the decidability of the model checking problem.

Specifications. NUXMV's support for LTL has been extended to allow for the use of $\text{MTL}_{0,\infty}$ operators [12] and other operators such as event-freezing functions [13] and dense version of LTL x and y operators. $\text{MTL}_{0,\infty}$ bounded operators extend the LTL ones of NUXMV to allow for bounds either of the form $[c, \infty)$, where c is a constant greater or equal to 0, e.g. $F[0, +\infty) \varphi$, or generic expressions over parametric/frozen variables: e.g. $F[0, 3+v] \varphi$ where v is a frozen variable.

In timed setting, next and previous operators come in two possible versions. The standard LTL operators x and y require to hold, respectively after and before, a discrete transition. Dually, x^\sim and y^\sim have been introduced to allow to predicate about the evolution over time of the system. They are always FALSE in discrete steps and hold in time elapses if the argument holds in the open interval immediately after/before (resp.) the current step. The disjunction $x(\varphi) \vee x^\sim(\varphi)$ allows to check if the argument φ holds after the current state without distinction between time or discrete evolution.

The event-freezing operators *at next* and *at last*, written $@F^\sim$ and $@O^\sim$, are binary operators allowed in LTL specifications. The left-hand side is a term, while the right-hand side is a temporal formula. They return the value of the term respectively at the next and at the last point in time in which the formula is true. If the formula will [has] never happen [happened] the operator evaluates to a default value.

`time_until` and `time_since` are two additional unary operators that can be used in LTL specifications of timed models. Their argument must be a Boolean predicate over current and next variables. `time_until`(φ) evaluates to the amount of time elapse required to reach the next state in which φ holds, while `time_since`(φ) evaluates to the amount of time elapsed from the last state in which φ held. As for the $@F^\sim$ and $@O^\sim$ operators if no such state exists they are assigned to a default value.

4 Extending Traces

Timed Traces. The semantics of NUXMV has been extended to take into account the timing aspects in case of super-dense time. While in the discrete time case, the execution trace is given by a sequence of states connected by discrete transitions (i.e., satisfying the transition condition), in the super-dense time case the execution trace is such that every pair of consecutive states is a discrete or a timed transition. As in the discrete case, discrete transitions are pair of states satisfying the transition condition. As in timed automata, in a timed transition time elapses for a certain amount (referred to as `delta_time`), clocks increase of the same amount, while discrete variables do not change.

Lasso-Shaped Traces with Diverging Variables. Traditionally, the only infinite paths supported by NUXMV have been those in lasso shape, i.e. those traces which can be

represented by a finite prefix s_0, s_1, \dots, s_l (called the stem) followed by a finite suffix $s_{l+1}, \dots, s_k \equiv s_l$ (called the loop), which can be repeated infinitely many times. While this representation is sufficient for finite-state systems (because in a finite-state setting if a system does not satisfy an LTL property, then a lasso-shaped counter-example trace is guaranteed to exist), this is an important limitation in an infinite-state context, in which lasso-shaped counter-examples are not guaranteed to exist. (As a simple example, consider a system $M := \langle \{x\}, (x = 0), (x' = x + 1) \rangle$ in which $x \in \mathbb{Z}$. Then $M \not\models \mathbf{GF}(x = 0)$, but clearly M has no lasso-shaped trace). In fact, this is especially relevant for timed transition systems, which, by the presence of the always-diverging variable *time*, admit no lasso-shaped trace.

In order to overcome this limitation, we introduce new kinds of infinite traces, which we call *lasso-shape traces with diverging variables* (to allow also for representing traces with variables whose value might be diverging). We modified the bounded model checking algorithms to leverage on this new representation to then extend the capabilities to find witnesses for a given property. This representation significantly extends the capabilities of NUXMV to find witnesses for violated LTL and MTL properties on timed transition systems (see experimental evaluation).

Definition 1. Let $\pi := s_0, s_1, \dots, s_l, \dots$ be an infinite trace of a system M over variables V . We say that π is a lasso-shaped trace with diverging variables iff there exist indexes $0 \leq l \leq k$, a partitioning of V into sets X and Y ($V = X \uplus Y$) and an expression $f_y(V)$ over V for every variable $y \in Y$ such that, for every $i > k$,

$$s_i(v) := \begin{cases} s_{l+((i-l) \bmod (k-l))}(v) & \text{if } v \in X \text{ (like in lasso-shaped traces);} \\ f_v(s_{i-1}) & \text{if } v \in Y \text{ (as function of previous state).} \end{cases}$$

Intuitively, the idea of lasso-shaped traces with diverging variables is to provide a finite representation for infinite traces that is more general than simple lasso-shaped ones, and which allows to capture more interesting behaviors of timed transition systems.

Example 1. Consider the system $M := \langle \{y, b\}, \neg b \wedge y = 0, (b' = \neg b) \wedge (b \rightarrow y' = y + 1) \wedge (\neg b \rightarrow y' = y) \rangle$. Then one lasso-shaped trace for M is given by: $\pi := s_0, s_1, s_2$, where $s_0 := \{b \mapsto \perp, y \mapsto 0\}$, $s_1 := \{b \mapsto \top, y \mapsto 0\}$, and $s_2 := \{b \mapsto \perp, y \mapsto 1\}$; the trace is lasso-shaped with diverging variables considering $Y := \{y\}$; the loop-back at index 0, and $f_y(b, y) := b ? y + 1 : y$.

Extended BMC for Traces with Divergent Clocks. The definition above requires the existence of the functions f_y for computing the updates of diverging variables. In case y is a clock variable, we can define a region $\llbracket \phi_y \rrbracket$ in which y can diverge (i.e., $f_y = y + \delta$, where δ is the delta time variable).

In order to capture lasso-shaped traces with diverging variables, we can modify the BMC encoding as follows. Let $\bigvee_{l=0}^k (\bigwedge_{v \in X \uplus Y} (v^l = v^k) \wedge {}_l \llbracket \varphi \rrbracket_k^0)$ be the formula

representing the BMC encoding of [14] at depth k with all possible loop-backs $0 \leq l \leq k$ for a given formula φ . The encoding is extended as follows:

$$\bigvee_{l=0}^k \left(\left(\bigwedge_{x \in X} (x^l = x^k) \wedge \bigwedge_{y \in Y} (y^l = y^k \vee \bigwedge_{i=l}^k [\![\phi_y]\!]_i) \right) \wedge {}_l[\![\varphi]\!]_k^0 \right).$$

The correctness of the encoding relies on a safe choice of the set Y , falling back to the incomplete lasso-shaped case when some syntactic restrictions on the expressions containing clocks are not met (see appendix for more details).

5 Related Work

There are many tools that allow for the specification and verification of infinite state symbolic synchronous transition systems. Given the focus of this paper, here we restrict our attention to tools supporting timed systems and/or MTL properties.

Uppaal [15], the reference tool for timed systems verification, supports only bounded variable types and therefore finite asynchronous TTS. Properties are limited to a subset of the branching-time logic TCTL [16, 17]. LTSmin [18] and Divine [19] are two model checkers that support the Uppaal specification language and properties specified in LTL. RTD-Finder [20] handles only safety properties for real-time component-based systems specified in RT-BIP. The verification is based on a compositional computation of an invariant over-approximating the set of reachable states of the system and leverages on counterexample-based invariant refinement algorithm. The ZOT Bounded Model/Satisfiability Checker [21] supports different logic languages through a multi-layered approach based on LTL with past operators. Similarly to NUXMV, ZOT supports dense-time MTL. It leverages only on SMT-based Bounded Model Checking, and is therefore unable to prove that properties hold. Atmoc [22] implements an extension of IC3 [7] and K-induction [23] to deal with symbolic timed transition systems. It supports both invariant and $MTL_{0,\infty}$ properties, although for the latter it only supports bounded model checking. CTAV [24] reduces the model checking problem for an $MTL_{0,\infty}$ property φ to a symbolic language emptiness check of a timed Büchi automata for φ .

Differently from all the above tools NUXMV is able to prove $MTL_{0,\infty}$ properties on timed transition systems with infinite domain variables.

6 Experimental Evaluation

We compared NUXMV with Atmoc [22], CTAV [24], ZOT [21], Divine [19], LTSmin [18], and Uppaal [25].

For the evaluation we considered (i) scalable benchmarks taken from competitor tools distributions and from the literature; (ii) handcrafted benchmarks to stress various language features. In particular, we considered different versions of the Fisher mutual exclusion protocol (correct and buggy) with different properties, different versions of the emergency diesel generator problem (previously studied with Atmoc [22]). Finally we considered also the validity checks of some MTL properties also taken from [22].

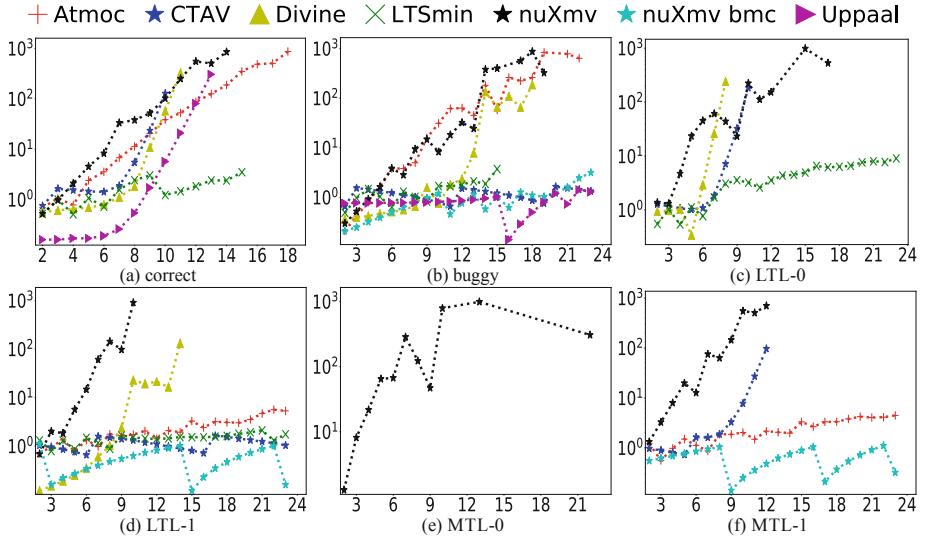


Fig. 3. Runtime for the Fisher mutual exclusion problem; x-axis: number of processes, y-axis: time (s). LTL-1 and MTL-1 properties are the bounded version of resp. LTL-0 and MTL-0.

We run all the experiments on a PC equipped with a 3.7 GHz Xeon quad core CPU and 16 Gb of RAM, using a time/memory limit of 1000 s/10 Gb for each test. We refer the reader to [26] to retrieve all the data to reproduce this experimental evaluation.

The results of the evaluation are reported in Fig. 3 for the Fisher family of experiments, and in Fig. 4 for the emergency diesel generator family of problems (CTAU does not appear in the plot of MTL-0 because it wrongly reports a counterexample although MTL-0 is the bounded version of LTL-0). While the results for the validity check of pure MTL properties are reported in Fig. 5. In the plots NUXMV refers to runtime for the IC3 with implicit abstraction in lockstep with BMC with the modified loop condition algorithm, and NUXMV-bmc refers to runtime for BMC alone with the modified loop condition algorithm. The results show that NUXMV is competitive with and in many cases performs better than other state-of-the-art tools, especially on validity problems for MTL_{0,∞}.

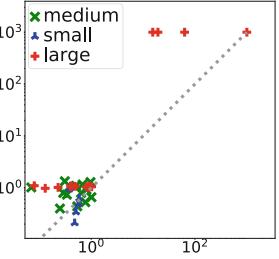


Fig. 4. Result for the runtime (s) for the emergency diesel generator family of problems: NUXMV (x) vs Atmoc (y).

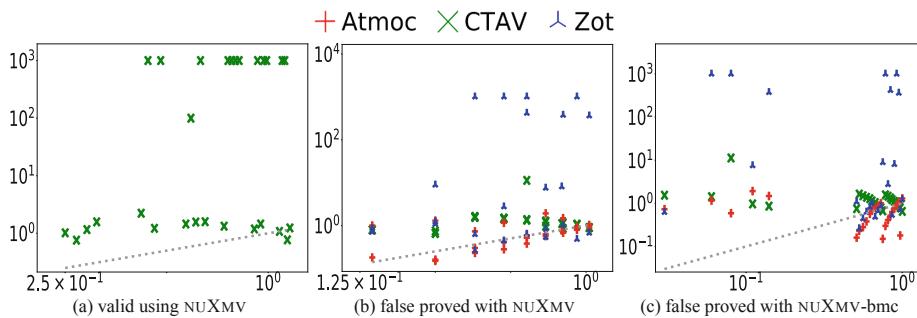


Fig. 5. Runtime (s) for the validity checks of MTL properties.

7 Conclusions

We presented the new version of NUXMV, a state-of-the art symbolic model checker for finite and infinite-state transition systems, that we extended to allow for the specification of synchronous timed transition systems and of $\text{MTL}_{0,\infty}$ properties. To support the new features, we extended the NUXMV language, we allowed for the specification $\text{MTL}_{0,\infty}$ formulas with parametric intervals, we adapted the model checking algorithms to find for lasso-shaped traces (over discrete semantics) where clocks may diverge. We evaluated the new features comparing NUXMV with other verification tools for timed automata, considering different benchmarks. The results show that NUXMV is competitive with and in many cases performs better than state-of-the-art tools, especially on validity problems for $\text{MTL}_{0,\infty}$.

References

1. Cavada, R., et al.: The NUXMV symbolic model checker. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 334–342. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_22
2. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT Solver. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 93–107. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_7
3. Koymans, R.: Specifying real-time properties with metric temporal logic. Real-Time Syst. 2(4), 255–299 (1990)
4. Ouaknine, J., Worrell, J.: On the decidability of metric temporal logic. In: Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science. LICS 2005, pp. 188–197. IEEE (2005)
5. Somenzi, F.: CUDD: Colorado University Decision Diagram package – release 2.4.1
6. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24605-3_37
7. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 70–87. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18275-4_7

8. Hassan, Z., Bradley, A.R., Somenzi, F.: Better generalization in IC3. In: FMCAD, pp. 157–164. IEEE (2013)
9. Vizel, Y., Grumberg, O., Shoham, S.: Lazy abstraction and sat-based reachability in hardware model checking. In: Cabodi, G., Singh, S. (eds.) FMCAD, pp. 173–181. IEEE (2012)
10. Claessen, K., Sörensson, N.: A liveness checking algorithm that counts. In: Cabodi, G., Singh, S. (eds.) FMCAD, pp. 52–59. IEEE (2012)
11. Schuppan, V., Biere, A.: Liveness checking as safety checking for infinite state spaces. *Electr. Notes Theor. Comput. Sci.* **149**(1), 79–96 (2006)
12. Alur, R., Feder, T., Henzinger, T.A.: The benefits of relaxing punctuality. *J. ACM* **43**(1), 116–146 (1996)
13. Tonetta, S.: Linear-time Temporal Logic with Event Freezing Functions. In: GandALF, pp. 195–209 (2017)
14. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. *Adv. Comput.* **58**, 117–148 (2003)
15. Behrmann, G., David, A., Larsen, K.G.: A tutorial on UPPAAL. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30080-9_7
16. Bouyer, P.: Model-checking timed temporal logics. In: Areces, C., Demri, S. (eds.) Proceedings of the 4th Workshop on Methods for Modalities (M4M-5). Electronic Notes in Theoretical Computer Science, vol. 1, pp. 323–341. Elsevier Science Publishers, Cachan, March 2009
17. Bouyer, P., Laroussinie, F., Markey, N., Ouaknine, J., Worrell, J.: Timed temporal logics. In: Aceto, L., Bacci, G., Bacci, G., Ingólfssdóttir, A., Legay, A., Mardare, R. (eds.) Models, Algorithms, Logics and Tools. LNCS, vol. 10460, pp. 211–230. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63121-9_11
18. Kant, G., Laarman, A., Meijer, J., van de Pol, J., Blom, S., van Dijk, T.: LTSmin: high-performance language-independent model checking. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 692–707. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_61
19. Baranová, Z., et al.: Model checking of C and C++ with DIVINE 4. In: D’Souza, D., Narayan Kumar, K. (eds.) ATVA 2017. LNCS, vol. 10482, pp. 201–207. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68167-2_14
20. Ben-Rayana, S., Bozga, M., Bensalem, S., Combaz, J.: RTD-finder: a tool for compositional verification of real-time component-based systems. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 394–406. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_23
21. Pradella, M.: A user’s guide to zot. CoRR abs/0912.5014 (2009)
22. Kindermann, R., Juntila, T.A., Niemelä, I.: Smt-based induction methods for timed systems. CoRR abs/1204.5639 (2012)
23. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In: Hunt, W.A., Johnson, S.D. (eds.) FMCAD 2000. LNCS, vol. 1954, pp. 127–144. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-40922-X_8
24. Li, G.: Checking timed büchi automata emptiness using LU-abstractions. In: Ouaknine, J., Vaandrager, F.W. (eds.) FORMATS 2009. LNCS, vol. 5813, pp. 228–242. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04368-0_18
25. Larsen, K.G., Lorber, F., Nielsen, B.: 20 years of UPPAAL enabled industrial model-based validation and beyond. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11247, pp. 212–229. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03427-6_18
26. Cimatti, A., Griggio, A., Magnago, E., Roveri, M., Tonetta, S.: Extending nuXmv with timed transition systems and timed temporal properties (extended version) (2019). Extended version with data to reproduce experiments <https://nuxmv.fbk.eu/papers/cav2019>

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Cerberus-BMC: A Principled Reference Semantics and Exploration Tool for Concurrent and Sequential C

Stella Lau^{1,2(✉)}, Victor B. F. Gomes²,
Kayvan Memarian², Jean Pichon-Pharabod²,
and Peter Sewell²

¹ MIT, Cambridge, USA
stellal@mit.edu

² University of Cambridge, Cambridge, UK
[{victor.gomes,kayvan.memarian,
jean.pichon-pharabod,peter.sewell}@cl.cam.ac.uk](mailto:{victor.gomes,kayvan.memarian,jean.pichon-pharabod,peter.sewell}@cl.cam.ac.uk)



Abstract. C remains central to our infrastructure, making verification of C code an essential and much-researched topic, but the semantics of C is remarkably complex, and important aspects of it are still unsettled, leaving programmers and verification tool builders on shaky ground. This paper describes a tool, Cerberus-BMC, that for the first time provides a principled reference semantics that simultaneously supports (1) a choice of concurrency memory model (including substantial fragments of the C11, RC11, and Linux kernel memory models), (2) a modern memory object model, and (3) a well-validated thread-local semantics for a large fragment of the language. The tool should be useful for C programmers, compiler writers, verification tool builders, and members of the C/C++ standards committees.

1 Introduction

C remains central to our infrastructure, widely used for security-critical components of hypervisors, operating systems, language runtimes, and embedded systems. This has prompted much research on the verification of C code, but the semantics of C is remarkably complex, and important aspects of it are still unsettled, leaving programmers and verification tool builders on shaky ground. Here we are concerned with three aspects:

1. *The Concurrency Memory Model.* The 2011 versions of the ISO C++ and C standards adopted a new concurrency model [3, 12, 13], formalised during the development process [11], but the model is still in flux: various fixes have been found to be necessary [9, 14, 26]; the model still suffers from the “thin-air problem” [10, 15, 35]; and Linux kernel C code uses a different model, itself recently partially formalised [7].

2. The Memory Object Model. A priori, one might imagine C follows one of two language-design extremes: a concrete byte-array model with pointers that are simply machine words, or an abstract model with pointers combining abstract block IDs and structured offsets. In fact C is neither of these: it permits casts between pointer and integer types, and manipulation of their byte representations, to support low-level systems programming, but, while at runtime a C pointer will typically just be a machine word, compiler analyses and optimisations reason about abstract notions of the provenance of pointers [27, 29, 31]. This is a subject of active discussion in the ISO C and C++ committees and in compiler development communities.

3. The Thread-Local Sequential Semantics. Here, there are many aspects, e.g. the loosely specified evaluation order, the semantics of integer promotions, many kinds of undefined behaviour, and so on, that are (given an expert reading) reasonably well-defined in the standard, but that are nonetheless very complex and widely misunderstood. The standard, being just a prose document, is not *executable as a test oracle*; it is not a reference semantics usable for exploration or automated testing.

Each of these is challenging in isolation, but there are also many subtle interactions between them. For example, between (1) and (3), the pre-C11 ISO standard text was in terms of sequential stepwise execution of an (informally specified) abstract machine, while the C11 concurrency model is expressed as a predicate over complete candidate executions, and the two have never been fully reconciled – e.g. in the standard’s treatment of object lifetimes. Then there are fundamental issues in combining the ISO treatment of undefined behaviour with that axiomatic-concurrency-model style [10, §7]. Between (1) and (2), one has to ask about the relationships between the definition of data race and the treatment of uninitialised memory and padding. Between (2) and (3), there are many choices for what the C memory object model should be, and how it should be integrated with the standard, which are currently under debate. Between all three one has to consider the relationships between uninitialised and thin-air values and the ISO notions of unspecified values and trap representations. These are all open questions in what the C semantics and ISO standard are (or should be). We do not solve them here, but we provide a necessary starting point: a tool embodying a precise reference semantics that lets one explore examples and debate the alternatives.

We describe a tool, Cerberus-BMC, that for the first time lets one explore the allowed behaviours of C test programs that involve all three of the above. It is available via a web interface at <http://cerberus.cl.cam.ac.uk/bmc.html>.

For (1), Cerberus-BMC is parameterised on an axiomatic memory concurrency model: it reads in a definition of the model in a Herd-like format [6], and so can be instantiated with (substantial fragments of) either the C11 [3, 9, 12–14], RC11 [26], or Linux kernel [7] memory models. The model can be edited in the web interface. Then the user can load (or edit in the web interface) a small C program. The tool first applies the Cerberus compositional translation (or elab-

oration) into a simple Core language, as in [29,31]; this elaboration addresses (3) by making many of the thread-local subtleties of C explicit, including the loose specification of evaluation order, arithmetic conversions, implementation-defined behaviour, and many kinds of undefined behaviour. Core computation is simply over mathematical integers, with explicit memory actions to interface with the concurrency and memory object models. However, there is a mismatch between the axiomatic style of the concurrency models for C (expressed as predicates on arbitrary candidate executions) with the operational style of the previous thread-local operational semantics for Core. We address this by replacing the latter with a new translation from Core into SMT problems. This is integrated with the concurrency model, also translated into SMT, following the ideas of [5]. These are furthermore integrated with an SMT version of parts of the PNVI (provenance-not-via-integers) memory object model of [29], the basis for ongoing work within the ISO WG14 C standards committee, addressing (2). The resulting SMT problems are passed to Z3 [32]. The web interface then provides a graphical view of the allowed concurrent executions for small test programs.

The Cerberus-BMC tool should be useful for programmers, compiler writers, verification tool builders, and members of the C/C++ standards committees. We emphasise that it is intended as an executable reference semantics for small test programs, not itself as a verification tool that can be applied to larger bodies of C: we have focussed on making it transparently based on principled semantics for all three aspects, without the complexities needed for a high-performance verification tool. But it should aid the construction of such.

Caveats and Limitations. Cerberus-BMC covers many features of 1–3, but far from all. With respect to the concurrency memory model, we support substantial fragments of the C11, RC11, and Linux kernel memory models. We omit locks and the (deprecated) C11/RC11 consume accesses. We only cover compare-exchange read-modify-write operations, and the fragment of RCU restricted to `read_rcu_lock()`, `read_rcu_unlock()`, and `synchronize_rcu()` used in a linear way, without control-flow-dependent calls to RCU, and without nesting.

With respect to the memory object model, we do not currently support dynamic allocation or manipulation of byte representations (such as with `char*` pointers), and we do not address issues such as subobject provenance (an open question within WG14).

With respect to the thread semantics, our translation to SMT does not currently cover arbitrary pointer type-casting, function pointers, multi-dimensional arrays, unions, floating point, bitwise operations, and variadic functions, and only covers simple structs. In addition, we inherit the limitations of the Cerberus thread semantics as per [29].

Related Work. There is substantial prior work on tools for concurrency semantics and for C semantics, but almost none that combines the two. On the concurrency semantics side, CppMem [1,11] is a web-interface tool that computes the allowed concurrent behaviours of small tests with respect to variants (now somewhat

outdated) of the C11 model, but it does not support other concurrency models or a memory object model, and it supports only a small fragment of C. Herd [6,8] is a command-line tool that computes the allowed concurrent behaviours of small tests with respect to arbitrary axiomatic concurrency models expressed in its `cat` language, but without a memory object model and for tests which essentially just comprise memory events, without a C semantics. MemAlloy [38] and MemSynth [16] also support reasoning about axiomatic concurrency models, but again not integrated with a C language semantics.

On the C semantics side, several projects address sequential C semantics but without concurrency. We build here on Cerberus [28, 29, 31], a web-interface tool that computes the allowed behaviours (interactively or exhaustively) for moderate-sized tests in a substantial fragment of sequential C, incorporating various memory object models (an early version supported Nienhuis's operational model for C11 concurrency [33], but that is no longer integrated). KCC and RV-Match [19, 21, 22] provide a command-line semantics tool for a substantial fragment of C, again without concurrency. Krebbers gives a Coq semantics for a somewhat smaller fragment [24].

Then there is another large body of work on model-checking tools for sequential and concurrent C. These are all optimised for model-checking performance, in contrast to the Cerberus-BMC emphasis on expressing the semantic envelope of allowed behaviour as clearly as we can (and, where possible, closely linked to the ISO standard). The former include tis-interpreter [18, 36], CBMC [17, 25], and ESBMC [20]. On the concurrent side, as already mentioned, we build on the approach of [5], which integrated various hardware memory concurrency models with CBMC. CDSChecker [34] supports something like the C/C++11 concurrency model, but subject to various limitations [34, §1.3]. It is implemented using a dynamically-linked shared library for the C and C++ atomic types, so implicitly adopts the C semantic choices of whichever compiler is used. RCMC [23], supports memory models that do not exhibit Load Buffering (LB), for an idealised thread-local language. Nidhugg [4] supports only hardware memory models: SC, TSO, PSO, and versions of POWER and ARM.

2 Examples

We now illustrate some of what Cerberus-BMC can do, by example.

Concurrency Models. First, for C11 concurrency, Fig. 1 shows a screenshot for a classic message-passing test, with non-atomic writes and reads of `x`, synchronised with release/acquire writes and reads of `y`. The test uses an explicit parallel composition, written `{ -{...} || ... } -`, to avoid the noise from the extra memory actions in `pthread_create`. The consistent race-free UB-free execution on the right shows the synchronisation working correctly: after the `i` read-acquire of `y=1`, the `l` non-atomic read of `x` has to read `x=1` (there are no consistent executions in which it does not). As usual in C/C++ candidate execution graphs, `rf` are reads-from edges, `sb` is sequenced-before (program order), `mo` is modification

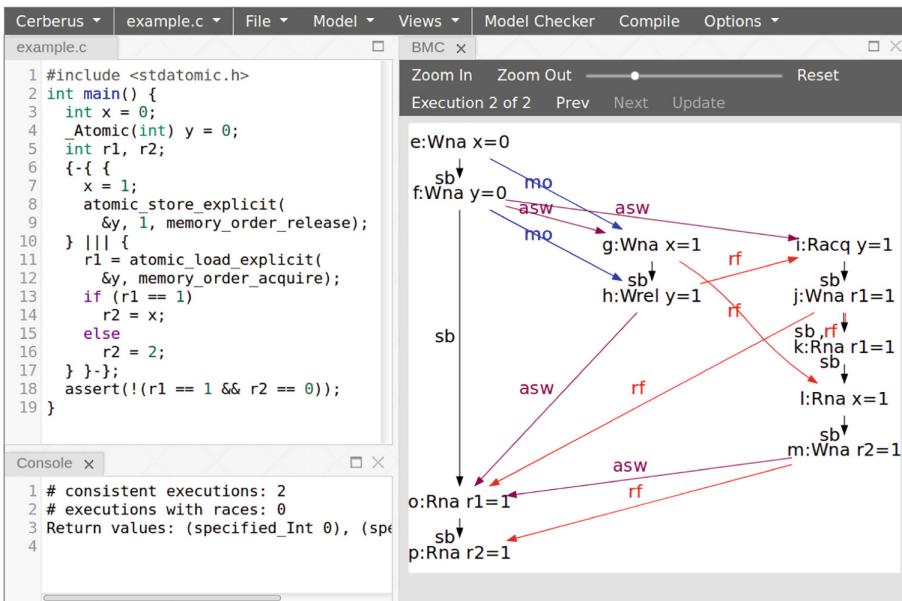


Fig. 1. Cerberus-BMC Screenshot: C11 Release/Acquire Message Passing. If the read of y is 1, then the last thread has to see the write of 1 to x .

```
#include "linux.h"
int main() {
    int x = 0, y = 0;
    int r1, r2;
    {-{
        WRITE_ONCE(x, 1);
        // synchronize_rcu();
        WRITE_ONCE(y, 1);
    } ||| {
        rcu_read_lock();
        r1 = READ_ONCE(x);
        r2 = READ_ONCE(y);
        rcu_read_unlock();
    } }-
    assert(!(r1==0&&r2==1));
}
```

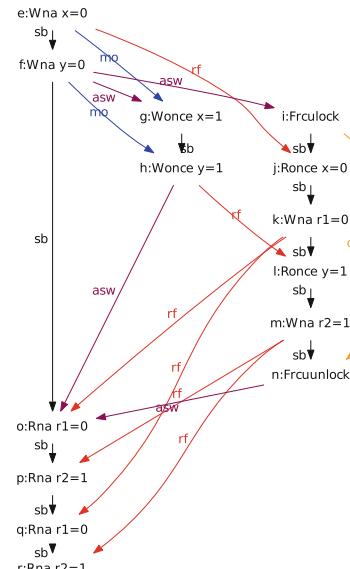


Fig. 2. Linux kernel memory model RCU lock. Without `synchronize_rcu()`, the reads of x and y can see 0 and 1 (as shown), even though they are enclosed in an RCU lock. With synchronization, after reading $x=1$, the last thread has to see $y=1$.

order (the coherence order between atomic writes to the same address), and `asw` is additional-synchronised-with, between parent and child threads and vice versa. Read and write events (R/W) are annotated `na` for non-atomic and `rel/acq` for release/acquire.

For the Linux kernel memory model, the example in Fig. 2 shows an RCU (read-copy-update) synchronisation.

Memory Object Model. The example below illustrates a case where one cannot assume that C has a concrete memory object model: pointer provenance matters. In some C implementations, `x` and `y` will happen to be allocated adjacent (the `__BMC_ASSUME` restricts attention to those executions). Then `&x+1` will have the same numeric address as `&y`, but the write `*p=11` is undefined behaviour rather than a write to `y`. This was informally described in the 2004 ISO WG14 C standards committee response to Defect Report 260 [37], but has never been incorporated into the standard itself. Cerberus-BMC correctly reports UB found: source.c:8:5-7, `UB043_indirection_invalid_value` following the PNVI (provenance-not-via-integers) memory object model of [29].

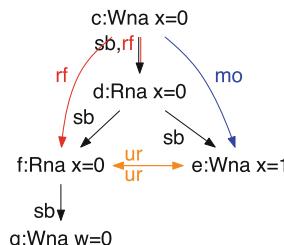
ISO Subtleties. Turning to areas where the ISO standard is clear to experts but widely misunderstood, in the example on the right ISO leaves it implementation-defined whether `char` is signed or unsigned. In the former case, the ISO integer promotion and conversion semantics will make the equality test false, leading to a division by 0, which is undefined behaviour.

The example below shows the correct treatment of the ISO standard's loose specification of evaluation order, together with detection of the concurrency model's *unsequenced races* (`ur` in the diagram): there are write and read accesses to `x` that are unrelated by sequenced-before (`sb`), and not otherwise synchronised and hence unrelated by happens-before, which makes this program undefined behaviour.

```
int main() {
    int x=0;
    int w;
    w = x++ + x;
}
```

```
#include <stdint.h>
int x = 1, y = 2;
int main() {
    int *p = &x + 1;
    int *q = &y;
    __BMC_ASSUME((intptr_t)p==(intptr_t)q);
    if ((intptr_t)p==(intptr_t)q)
        *p = 11; // does this have UB?
}
```

```
int main() {
    char c1 = 0xff;
    unsigned char c2 = 0xff;
    return 1 / (c1 == c2);
}
```



Treiber Stack. Finally, demonstrating the combination of all three aspects, we implemented a modified Treiber stack (the `push()` function is shown in Fig. 3) with relaxed accesses to struct fields. Although the Treiber stack is traditionally implemented by spinning on a compare-and-swap, as that can spin unboundedly, we instead use `__BMC_ASSUME` to restrict executions to those where the compare-and-swap succeed. Our tool correctly detects the different results from the concurrent relaxed-memory execution of threads concurrently executing the `push` and `pop` functions.

```

struct Node { int data; struct Node *next; };
struct Node * __Atomic T;
void push(struct Node **x, int v) {
    struct Node *t;
    x->data = v;
    t = atomic_load_explicit(&T, memory_order_relaxed);
    x->next = t;
    __BMC_ASSUME(atomic_compare_exchange_strong_explicit(&T, &t, x,
        memory_order_acq_rel, memory_order_relaxed));
}

```

Fig. 3. Treiber stack `push()`

```

1 proc main (): eff loaded integer :=
2   let strong x: pointer = create(Ivalignof('signed int'), 'signed int') in
3   let strong a_437: loaded integer = pure(Specified(1)) in
4   store('signed int', x, conv_loaded_int('signed int', a_437)) ;
5   kill(x) ;
6   (save ret_435: loaded integer (a_436: loaded integer:= Specified(0)) in
7     pure(a_436))

```

Fig. 4. Core program corresponding to `int main(){int x = 1}`. Core is essentially a typed, first-order lambda calculus with explicit memory actions such as `create` and `store` to interface with the concurrency and memory object models.

3 Implementation

After translating a C program into Core (see Fig. 4), Cerberus-BMC does a sequence of Core-to-Core rewrites in the style of bounded model checkers such as CBMC: it unwinds loops and inlines function calls (to a given bound), and renames symbols to generate an SSA-style program.

The explicit representation of memory operations in Core as first-order constructs allows the SMT translation to be easily separated into three components: the translation from Core to SMT, the memory object model constraints, and the concurrency model constraints.

1. Core to SMT. Each value in Core is represented as an SMT expression, with fresh SMT constants for memory actions such as `create` and `store` (e.g. lines 2 and 4), the concrete values of which are constrained by the memory object and concurrency models. The elaboration of C to Core makes thread-local undefined behaviour (as opposed to undefined behaviour from concurrency or memory layout), like signed integer overflow, explicit with a primitive `undef` construct. Undefined behaviour is then encoded in SMT as reachability of `undef` expressions, that is, satisfiability of the control-flow guards up to them.

2. Memory Object Model. As in the PNVI semantics [30], Cerberus-BMC represents pointers as pairs (π, a) of a provenance π and an integer address a . The provenance of a pointer is taken into account when doing memory accesses, pointer comparisons, and casts between integer and pointer values. Our tool models address allocation nondeterminism by constraining address values based on allocations to be appropriately aligned and non-overlapping, but not constraining the addresses otherwise.

3. Concurrency Model. Cerberus-BMC statically extracts memory actions and computes an extended pre-execution containing relations such as program order. As control flow can not be statically determined, memory actions are associated with an SMT boolean guard representing the control flow conditions upon which the memory action is executed.

Cerberus-BMC reads in a model definition in a subset of the herd `cat` language large enough to express C11, RC11, and Linux, and generates a set of quantifier-free SMT expressions corresponding to the model's constraints on relations. These constraints are based on a set of "built-in" relations defined in SMT such as `rf`. Cerberus-BMC then queries Z3 to extract all the executions, displaying the load/store values and computed relations for the user.

4 Validation

We validate correctness of the three aspects of Cerberus-BMC as follows, though, as ever, additional testing would be desirable. Performance data, demonstrating practical usability, is from a MacBook Pro 2.9 GHz Intel Core i5.

For C11 and RC11 concurrency, we check on 12 classic litmus tests. For Linux kernel concurrency, we hand-translated the 9 non-RCU tests and 4 of the RCU tests of [7] into C, and automatically translated the 40 tests of [2]. Running all the non-RCU tests takes less than 5 min; the RCU tests are slower, of the order of one hour, perhaps because of the recursive definitions involved.

For the memory object model, we take the supported subset (36 tests) of the provenance semantics test suite of [29]. These single-threaded tests each run in less than a second.

For the thread-local semantics, the Cerberus pipeline to Core has previously been validated using GCC Torture, Toyota ITC, KCC, and Csmith-generated test suites [29]. We check the mapping to BMC using 50 hand-written tests and

the supported subset (400 tests) of the Toyota ITC test suite, each running in less than two minutes.

These test suites and the examples in the paper can be accessed via the CAV 2019 pop-up in the File menu of the tool.

Acknowledgments. This work was partially supported by EPSRC grant EP/K008528/1 (REMS), ERC Advanced Grant ELVER 789108, and an MIT EECS Graduate Alumni Fellowship.

References

1. CppMem: Interactive C/C++ memory model. <http://svr-pes20-cppmem.cl.cam.ac.uk/cppmem/index.html>
2. Litmus tests for validation LISA-language Linux-kernel memory models. <https://github.com/paulmckreua/litmus/tree/master/manual/lwn573436>
3. Programming Languages — C: ISO/IEC 9899:2011 (2011). A non-final but recent version is available at <http://www.open-std.org/jtc1/sc22/wg14/docs/n1539.pdf>
4. Abdulla, P.A., Aronis, S., Atig, M.F., Jonsson, B., Leonardsson, C., Sagonas, K.: Stateless model checking for TSO and PSO. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 353–367. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_28
5. Alglave, J., Kroening, D., Tautschnig, M.: Partial orders for efficient bounded model checking of concurrent software. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 141–157. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_9
6. Alglave, J., Maranget, L.: Herd7 (in the diy tool suite) (2015). <http://diy.inria.fr/>
7. Alglave, J., Maranget, L., McKenney, P.E., Parri, A., Stern, A.S.: Frightening small children and disconcerting grown-ups: concurrency in the Linux kernel. In: Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, 24–28 March 2018, pp. 405–418 (2018). <https://doi.org/10.1145/3173162.3177156>
8. Alglave, J., Maranget, L., Tautschnig, M.: Herding cats: modelling, simulation, testing, and data mining for weak memory. ACM TOPLAS **36**(2), 7:1–7:74 (2014)
9. Batty, M., Donaldson, A.F., Wickerson, J.: Overhauling SC atomics in C11 and OpenCl. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, 20–22 January 2016, pp. 634–648 (2016). <https://doi.org/10.1145/2837614.2837637>
10. Batty, M., Memarian, K., Nienhuis, K., Pichon-Pharabod, J., Sewell, P.: The problem of programming language concurrency semantics. In: Vitek, J. (ed.) ESOP 2015. LNCS, vol. 9032, pp. 283–307. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46669-8_12
11. Batty, M., Owens, S., Sarkar, S., Sewell, P., Weber, T.: Mathematizing C++ concurrency. In: Proceeding POPL (2011)
12. Becker, P. (ed.): Programming Languages — C++, iSO/IEC 14882:2011 (2011). A non-final but recent version is available at <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf>
13. Boehm, H.J., Adve, S.V.: Foundations of the C++ concurrency memory model. In: Proceedings of PLDI, pp. 68–78. ACM, New York (2008)

14. Boehm, H.J., Giroux, O., Vafeiadis, V.: P0668R2: Revising the C++ memory model. ISO WG21 paper (2018). <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0668r2.html>
15. Boehm, H., Demsky, B.: Outlawing ghosts: avoiding out-of-thin-air results. In: Proceedings of the Workshop on Memory Systems Performance and Correctness, MSPC 2014, Edinburgh, United Kingdom, 13 June 2014, pp. 7:1–7:6 (2014). <https://doi.org/10.1145/2618128.2618134>
16. Bornholt, J., Torlak, E.: Synthesizing memory models from framework sketches and litmus tests. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, 18–23 June 2017, pp. 467–481 (2017). <https://doi.org/10.1145/3062341.3062353>
17. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24730-2_15
18. Cuoq, P., Runarvot, L., Cherepanov, A.: Detecting strict aliasing violations in the wild. In: Bouajjani, A., Monniaux, D. (eds.) VMCAI 2017. LNCS, vol. 10145, pp. 14–33. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-52234-0_2
19. Ellison, C., Roșu, G.: An executable formal semantics of C with applications. In: Proceedings of POPL (2012)
20. Gadelha, M.Y.R., Monteiro, F.R., Morse, J., Cordeiro, L.C., Fischer, B., Nicole, D.A.: ESBMC 5.0: an industrial-strength C model checker. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, 3–7 September 2018, pp. 888–891 (2018)
21. Guth, D., Hathhorn, C., Saxena, M., Roșu, G.: RV-Match: practical semantics-based program analysis. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016, Part I. LNCS, vol. 9779, pp. 447–453. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41528-4_24
22. Hathhorn, C., Ellison, C., Rosu, G.: Defining the undefinedness of C. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, 15–17 June 2015, pp. 336–345 (2015). <https://doi.org/10.1145/2737924.2737979>
23. Kokologiannakis, M., Lahav, O., Sagonas, K., Vafeiadis, V.: Effective stateless model checking for C/C++ concurrency. PACMPL 2(POPL), 17:1–17:32 (2018). <https://doi.org/10.1145/3158105>
24. Krebbers, R.: The C standard formalized in CoQ. Ph.D. thesis, Radboud University Nijmegen, December 2015
25. Kroening, D., Tautschnig, M.: CBMC – C bounded model checker (competition contribution). In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 389–391. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_26
26. Lahav, O., Vafeiadis, V., Kang, J., Hur, C., Dreyer, D.: Repairing sequential consistency in C/C++11. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, 18–23 June 2017, pp. 618–632 (2017). <https://doi.org/10.1145/3062341.3062352>
27. Lee, J., Hur, C.K., Jung, R., Liu, Z., Regehr, J., Lopes, N.P.: Reconciling high-level optimizations and low-level code with twin memory allocation. In: Proceedings of the 2018 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2018, part of SPLASH 2018, Boston, MA, USA, 4–9 November 2018. ACM (2018)
28. Memarian, K., Gomes, V., Sewell, P.: Cerberus (2018). <http://cerberus.cl.cam.ac.uk/cerberus>

29. Memarian, K., et al.: Exploring C semantics and pointer provenance. In: Proceedings of 46th ACM SIGPLAN Symposium on Principles of Programming Languages, January 2019. Proc. ACM Program. Lang. 3, POPL, Article 67
30. Memarian, K., et al.: Exploring C semantics and pointer provenance. PACMPL 3(POPL), 67:1–67:32 (2019). <https://dl.acm.org/citation.cfm?id=3290380>
31. Memarian, K., et al.: Into the depths of C: elaborating the de facto standards. In: PLDI 2016: 37th Annual ACM SIGPLAN Conference on Programming Language Design and Implementation (Santa Barbara), June 2016. <http://www.cl.cam.ac.uk/users/pes20/cherberus/pldi16.pdf>. PLDI 2016 Distinguished Paper award
32. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
33. Nienhuis, K., Memarian, K., Sewell, P.: An operational semantics for C/C++11 concurrency. In: Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. ACM, New York (2016). <https://doi.org/10.1145/2983990.2983997>
34. Norris, B., Demsky, B.: CDSchecker: checking concurrent data structures written with C/C++ atomics. In: Proceedings of OOPSLA (2013)
35. Ou, P., Demsky, B.: Towards understanding the costs of avoiding out-of-thin-air results. PACMPL 2(OOPSLA), 136:1–136:29 (2018). <https://doi.org/10.1145/3276506>
36. TrustInSoft: `tis-interpreter` (2017). <http://trust-in-soft.com/tis-interpreter/>. Accessed 11 Nov 2017
37. WG14: Defect report 260, September 2004. http://www.open-std.org/jtc1/sc22/wg14/www/docs/dr_260.htm
38. Wickerson, J., Batty, M., Sorensen, T., Constantinides, G.A.: Automatically comparing memory consistency models. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, pp. 190–204. ACM, New York (2017). <https://doi.org/10.1145/3009837.3009838>

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.



Cyber-Physical Systems and Machine Learning



Multi-armed Bandits for Boolean Connectives in Hybrid System Falsification

Zhenya Zhang^{1,2} , Ichiro Hasuo^{1,2} ,
and Paolo Arcaini¹



¹ National Institute of Informatics, Tokyo, Japan

{zhangzy, hasuo, arcaini}@nii.ac.jp

² SOKENDAI (The Graduate University for Advanced Studies),
Hayama, Japan

Abstract. *Hybrid system falsification* is an actively studied topic, as a scalable quality assurance methodology for real-world cyber-physical systems. In falsification, one employs stochastic hill-climbing optimization to quickly find a counterexample input to a black-box system model. Quantitative *robust semantics* is the technical key that enables use of such optimization. In this paper, we tackle the so-called *scale problem* regarding Boolean connectives that is widely recognized in the community: quantities of different scales (such as speed [km/h] vs. rpm, or worse, rph) can mask each other's contribution to robustness. Our solution consists of integration of the *multi-armed bandit* algorithms in hill climbing-guided falsification frameworks, with a technical novelty of a new reward notion that we call *hill-climbing gain*. Our experiments show our approach's robustness under the change of scales, and that it outperforms a state-of-the-art falsification tool.

1 Introduction

Hybrid System Falsification. Quality assurance of *cyber-physical systems (CPS)* is attracting growing attention from both academia and industry, not only because it is challenging and scientifically interesting, but also due to the safety-critical nature of many CPS. The combination of physical systems (with continuous dynamics) and digital controllers (that are inherently discrete) is referred to as *hybrid systems*, capturing an important aspect of CPS. To verify hybrid systems is intrinsically hard, because the continuous dynamics therein leads to infinite search spaces.

More researchers and practitioners are therefore turning to *optimization-based falsification* as a quality assurance measure for CPS. The problem is formalized as follows.

The authors are supported by ERATO HASUO Metamathematics for Systems Design Project (No. JPMJER1603), JST.

© The Author(s) 2019

I. Dillig and S. Tasiran (Eds.): CAV 2019, LNCS 11561, pp. 401–420, 2019.

https://doi.org/10.1007/978-3-030-25540-4_23

The falsification problem

- Given: a model \mathcal{M} (that takes an input signal \mathbf{u} and

yields an output signal $\mathcal{M}(\mathbf{u})$), and a specification φ (a $\xrightarrow{\mathbf{u}}$ $\boxed{\mathcal{M}} \xrightarrow{\mathcal{M}(\mathbf{u})} \frac{\varphi}{\models \varphi ?}$)
temporal formula)

- Find: a *falsifying input*, that is, an input signal \mathbf{u} such that the corresponding output $\mathcal{M}(\mathbf{u})$ violates φ

In optimization-based falsification, the above problem is turned into an optimization problem. It is *robust semantics* of temporal formulas [12, 17] that makes it possible. Instead of the Boolean satisfaction relation $\mathbf{v} \models \varphi$, robust semantics assigns a quantity $[\mathbf{v}, \varphi] \in \mathbb{R} \cup \{\infty, -\infty\}$ that tells us, not only whether φ is true or not (by the sign), but also *how robustly* the formula is true or false. This allows one to employ hill-climbing optimization: we iteratively generate input signals, in the direction of decreasing robustness, hoping that eventually we hit negative robustness.

Table 1. Boolean satisfaction $\mathbf{w} \models \varphi$, and quantitative robustness values $[\mathbf{w}, \varphi]$, of three signals of speed for the STL formula $\varphi \equiv \square_{[0,30]}(\text{speed} < 120)$

signal \mathbf{w}	$\mathbf{w} \models \varphi$	$[\mathbf{w}, \varphi]$
	True	30
	True	10
	False	-10

An illustration of robust semantics is in Table 1. We use *signal temporal logic (STL)* [12], a temporal logic that is commonly used in hybrid system specification. The specification says the speed must always be below 120 during the time interval $[0, 30]$. In the search of an input signal \mathbf{u} (e.g. of throttle and brake) whose corresponding output $\mathcal{M}(\mathbf{u})$ violates the specification, the quantitative robustness $[\mathcal{M}(\mathbf{u}), \varphi]$ gives much more information than the Boolean satisfaction $\mathcal{M}(\mathbf{u}) \models \varphi$. Indeed, in Table 1, while Boolean satisfaction fails to discriminate the first two signals, the quantitative robustness indicates a tendency that the second signal is closer to violation of the specification.

In the falsification literature, stochastic algorithms are used for hill-climbing optimization. Examples include simulated annealing (SA), globalized Nelder-Mead (GNM [30]) and covariance matrix adaptation evolution strategy (CMA-ES [6]). Note that the system model \mathcal{M} can be black-box: we have only to observe the correspondence between input \mathbf{u} and output $\mathcal{M}(\mathbf{u})$. Observing an error $\mathcal{M}(\mathbf{u}')$ for some input \mathbf{u}' is sufficient evidence for a system designer to know that the system needs improvement. Besides these practical advantages, optimization-based falsification is an interesting scientific topic: it combines two different worlds of formal reasoning and stochastic optimization.

Optimization-based falsification started in [17] and has been developed vigorously [1, 3–5, 9, 11–13, 15, 27, 28, 34, 36, 38]. See [26] for a survey. There are mature tools such as Breach [11] and S-Taliro [5]; they work with industry-standard Simulink models.

Challenge: The Scale Problem in Boolean Superposition. In the field of hybrid falsification—and more generally in search-based testing—the following problem is widely recognized. We shall call the problem *the scale problem (in Boolean superposition)*.

Consider an STL specification $\varphi \equiv \square_{[0,30]}(\neg(rpm > 4000) \vee (speed > 20))$ for a car; it is equivalent to $\square_{[0,30]}((rpm > 4000) \rightarrow (speed > 20))$ and says that the speed should not be too small whenever the rpm is over 4000. According to the usual definition in the literature [11, 17], the Boolean connectives \neg and \vee are interpreted by $-$ and the supremum \sqcup , respectively; and the “always” operator $\square_{[0,30]}$ is by infimum \sqcup . Therefore the robust semantics of φ under the signal $(rpm, speed)$, where $rpm, speed : [0, 30] \rightarrow \mathbb{R}$, is given as follows.

$$\llbracket (rpm, speed), \varphi \rrbracket = \sqcap_{t \in [0,30]} ((4000 - rpm(t)) \sqcup (speed(t) - 20)) \quad (1)$$

A problem is that, in the supremum of two real values in (1), one component can totally *mask* the contribution of the other. In this specific example, the former (*rpm*) component can have values as big as thousands, while the latter (*speed*) component will be in the order of tens. This means that in hill-climbing optimization it is hard to use the information of both signals, as one will be masked.

Another related problem is that the efficiency of a falsification algorithm would depend on the choice of units of measure. Imagine replacing *rpm* with *rph* in (1), which makes the constant 4000 into 240000, and make the situation even worse.

These problems—that we call the *scale problem*—occur in many falsification examples, specifically when a specification involves Boolean connectives. We do need Boolean connectives in specifications: for example, many real-world specifications in industry are of the form $\square_I(\varphi_1 \rightarrow \varphi_2)$, requiring that an event φ_1 triggers a counter-measure φ_2 all the time.

One could use different operators for interpreting Boolean connectives. For example, in [21], \vee and \wedge are interpreted by $+$ and \times over \mathbb{R} , respectively. However, these choices do not resolve the scale problem, either. In general, it does not seem easy to come up with a fixed set of operators over \mathbb{R} that interpret Boolean connectives and are free from the scale problem.

Contribution: Integrating Multi-Armed Bandits into Optimization-Based Falsification. As a solution to the scale problem in Boolean superposition that we just described, we introduce a new approach that does *not* superpose robustness values. Instead, we integrate *multi-armed bandits (MAB)* in the existing framework of falsification guided by hill-climbing optimization.



Fig. 1. A multi-armed bandit for falsifying $\square_I(\varphi_1 \wedge \varphi_2)$

The MAB problem is a prototypical reinforcement learning problem: a gambler sits in front of a row of slot machines; their performance (i.e. average reward) is not known; the gambler plays a machine in each round and he continues with many rounds; and the goal is to optimize cumulative rewards. The gambler needs to play different machines and figure out their performance, at the cost of the loss of opportunities in the form of playing suboptimal machines.

In this paper, we focus on specifications of the form $\square_I(\varphi_1 \wedge \varphi_2)$ and $\square_I(\varphi_1 \vee \varphi_2)$; we call them (*conjunctive/disjunctive safety properties*). We identify an instance of the MAB problem in the choice of the formula (out of φ_1, φ_2) to try to falsify by hill climbing. See Fig. 1. We combine MAB algorithms (such as ε -greedy and UCB1, see Sect. 3.2) with hill-climbing optimization, for the purpose of coping with the scale problem in Boolean superposition. This combination is made possible by introducing a novel reward notion for MAB, called *hill-climbing gain*, that is tailored for this purpose.

We have implemented our MAB-based falsification framework in MATLAB, building on Breach [11].¹ Our experiments with benchmarks from [7, 24, 25] demonstrate that our MAB-based approach is a viable one against the scale problem. In particular, our approach is observed to be (almost totally) robust under the change of scaling (i.e. changing units of measure, such as from rpm to rph that we discussed after the formula (1)). Moreover, for the benchmarks taken from the previous works—they do not suffer much from the scale problem—our algorithm performs better than the state-of-the-art falsification tool Breach [11].

Related Work. Besides those we mentioned, we shall discuss some related works.

Formal verification approaches to correctness of hybrid systems employ a wide range of techniques, including model checking, theorem proving, rigorous numerics, nonstandard analysis, and so on [8, 14, 18, 20, 22, 23, 29, 32]. These are currently not very successful in dealing with complex real-world systems, due to issues like scalability and black-box components.

Our use of MAB in falsification exemplifies the role of the *exploration-exploitation trade-off*, the core problem in reinforcement learning. The trade-off has been already discussed for the verification of quantitative properties (e.g., [33]) and also in some works on falsification. A recent example is [36], where they use Monte Carlo tree search to force systematic exploration of the space of input signals. Besides MCTS, *Gaussian process learning (GP learning)* has also attracted attention in machine learning as a clean way of balancing exploitation and exploration. The GP-UCB algorithm is a widely used strategy there. Its use in hybrid system falsification is pursued e.g. in [3, 34].

More generally, *coverage-guided falsification* [1, 9, 13, 28] aims at coping with the exploration-exploitation trade-off. One can set the current work in this context—the difference is that we force systematic exploration on the specification side, not in the input space.

There have been efforts to enhance expressiveness of MTL and STL, so that engineers can express richer intentions—such as time robustness and frequency—in speci-

¹ Code obtained at <https://github.com/decyphir/breach>.

fications [2, 31]. This research direction is orthogonal to ours; we plan to investigate the use of such logics in our current framework.

A similar masking problem around Boolean connectives is discussed in [10, 19]. Compared to those approaches, our technique does not need the explicit declaration of *input vacuity* and *output robustness*, but it relies on the “hill-climbing gain” reward to learn the significance of each signal.

Finally, the interest in the use of deep neural networks is rising in the field of falsification (as well as in many other fields). See e.g. [4, 27].

2 Preliminaries: Hill Climbing-Guided Falsification

We review a well-adopted methodology for hybrid system falsification, namely the one guided by hill-climbing optimization. It makes essential use of quantitative *robust semantics* of temporal formulas, which we review too.

2.1 Robust Semantics for STL

Our definitions here are taken from [12, 17].

Definition 1 ((time-bounded) signal). Let $T \in \mathbb{R}_+$ be a positive real. An M -dimensional signal with a time horizon T is a function $\mathbf{w}: [0, T] \rightarrow \mathbb{R}^M$.

Let $\mathbf{w}: [0, T] \rightarrow \mathbb{R}^M$ and $\mathbf{w}': [0, T'] \rightarrow \mathbb{R}^M$ be M -dimensional signals. Their concatenation $\mathbf{w} \cdot \mathbf{w}': [0, T + T'] \rightarrow \mathbb{R}^M$ is the M -dimensional signal defined by $(\mathbf{w} \cdot \mathbf{w}')(t) = \mathbf{w}(t)$ if $t \in [0, T]$, and $(\mathbf{w} \cdot \mathbf{w}')(t) = \mathbf{w}'(t - T)$ if $t \in (T, T + T']$.

Let $0 < T_1 < T_2 \leq T$. The restriction $\mathbf{w}|_{[T_1, T_2]}: [0, T_2 - T_1] \rightarrow \mathbb{R}^M$ of $\mathbf{w}: [0, T] \rightarrow \mathbb{R}^M$ to the interval $[T_1, T_2]$ is defined by $(\mathbf{w}|_{[T_1, T_2]})(t) = \mathbf{w}(T_1 + t)$.

One main advantage of optimization-based falsification is that a system model can be a black box—observing the correspondence between input and output suffices. We therefore define a system model simply as a function.

Definition 2 (system model \mathcal{M}). A system model, with M -dimensional input and N -dim. output, is a function \mathcal{M} that takes an input signal $\mathbf{u}: [0, T] \rightarrow \mathbb{R}^M$ and returns a signal $\mathcal{M}(\mathbf{u}): [0, T] \rightarrow \mathbb{R}^N$. Here the common time horizon $T \in \mathbb{R}_+$ is arbitrary. Furthermore, we impose the following *causality* condition on \mathcal{M} : for any time-bounded signals $\mathbf{u}: [0, T] \rightarrow \mathbb{R}^M$ and $\mathbf{u}': [0, T'] \rightarrow \mathbb{R}^M$, we require that $\mathcal{M}(\mathbf{u} \cdot \mathbf{u}')|_{[0, T]} = \mathcal{M}(\mathbf{u})$.

Definition 3 (STL syntax). We fix a set \mathbf{Var} of variables. In STL, *atomic propositions* and *formulas* are defined as follows, respectively: $\alpha ::= f(x_1, \dots, x_N) > 0$, and $\varphi ::= \alpha \mid \perp \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \mathcal{U}_I \varphi$. Here f is an N -ary function $f: \mathbb{R}^N \rightarrow \mathbb{R}$, $x_1, \dots, x_N \in \mathbf{Var}$, and I is a closed non-singular interval in $\mathbb{R}_{\geq 0}$, i.e. $I = [a, b]$ or $[a, \infty)$ where $a, b \in \mathbb{R}$ and $a < b$.

We omit subscripts I for temporal operators if $I = [0, \infty)$. Other common connectives such as \rightarrow , \top , \square_I (always) and \diamond_I (eventually), are introduced as abbreviations: $\diamond_I \varphi \equiv \top \mathcal{U}_I \varphi$ and $\square_I \varphi \equiv \neg \diamond_I \neg \varphi$. An atomic formula $f(\mathbf{x}) \leq c$, where $c \in \mathbb{R}$, is accommodated using \neg and the function $f'(\mathbf{x}) := f(\mathbf{x}) - c$.

Definition 4 (robust semantics [12]). Let $\mathbf{w}: [0, T] \rightarrow \mathbb{R}^N$ be an N -dimensional signal, and $t \in [0, T]$. The t -shift of \mathbf{w} , denoted by \mathbf{w}^t , is the time-bounded signal $\mathbf{w}^t: [0, T-t] \rightarrow \mathbb{R}^N$ defined by $\mathbf{w}^t(t') := \mathbf{w}(t+t')$.

Let $\mathbf{w}: [0, T] \rightarrow \mathbb{R}^{|\text{Var}|}$ be a signal, and φ be an STL formula. We define the robustness $\llbracket \mathbf{w}, \varphi \rrbracket \in \mathbb{R} \cup \{\infty, -\infty\}$ as follows, by induction on the construction of formulas. Here \sqcap and \sqcup denote infimums and supremums of real numbers, respectively. Their binary version \sqcap and \sqcup denote minimum and maximum.

$$\begin{aligned}\llbracket \mathbf{w}, f(x_1, \dots, x_n) > 0 \rrbracket &:= f(\mathbf{w}(0)(x_1), \dots, \mathbf{w}(0)(x_n)) \\ \llbracket \mathbf{w}, \perp \rrbracket &:= -\infty \quad \llbracket \mathbf{w}, \neg\varphi \rrbracket := -\llbracket \mathbf{w}, \varphi \rrbracket \\ \llbracket \mathbf{w}, \varphi_1 \wedge \varphi_2 \rrbracket &:= \llbracket \mathbf{w}, \varphi_1 \rrbracket \sqcap \llbracket \mathbf{w}, \varphi_2 \rrbracket \quad \llbracket \mathbf{w}, \varphi_1 \vee \varphi_2 \rrbracket := \llbracket \mathbf{w}, \varphi_1 \rrbracket \sqcup \llbracket \mathbf{w}, \varphi_2 \rrbracket \\ \llbracket \mathbf{w}, \varphi_1 \mathcal{U}_I \varphi_2 \rrbracket &:= \sqcup_{t \in I \cap [0, T]} (\llbracket \mathbf{w}^t, \varphi_2 \rrbracket \sqcap \sqcap_{t' \in [0, t)} \llbracket \mathbf{w}^{t'}, \varphi_1 \rrbracket)\end{aligned}\tag{2}$$

For atomic formulas, $\llbracket \mathbf{w}, f(\mathbf{x}) > c \rrbracket$ stands for the vertical margin $f(\mathbf{x}) - c$ for the signal \mathbf{w} at time 0. A negative robustness value indicates how far the formula is from being true. It follows from the definition that the robustness for the eventually modality is given by $\llbracket \mathbf{w}, \Diamond_{[a, b]}(x > 0) \rrbracket = \sqcup_{t \in [a, b] \cap [0, T]} \mathbf{w}(t)(x)$.

The above robustness notion taken from [12] is therefore *spatial*. Other robustness notions take *temporal* aspects into account, too, such as “how long before the deadline the required event occurs”. See e.g. [2, 12]. Our choice of spatial robustness in this paper is for the sake of simplicity, and is thus not essential.

The original semantics of STL is Boolean, given as usual by a binary relation \models between signals and formulas. The robust semantics refines the Boolean one in the following sense: $\llbracket \mathbf{w}, \varphi \rrbracket > 0$ implies $\mathbf{w} \models \varphi$, and $\llbracket \mathbf{w}, \varphi \rrbracket < 0$ implies $\mathbf{w} \not\models \varphi$, see [17, Prop. 16]. Optimization-based falsification via robust semantics hinges on this refinement.

2.2 Hill Climbing-Guided Falsification

As we discussed in the introduction, the falsification problem attracts growing industrial and academic attention. Its solution methodology by hill-climbing optimization is an established field, too: see [1, 3, 5, 9, 11–13, 15, 26, 28, 34, 38] and the tools Breach [11] and S-TaLiRo [5]. We formulate the problem and the methodology, for later use in describing our multi-armed bandit-based algorithm.

Definition 5 (falsifying input). Let \mathcal{M} be a system model, and φ be an STL formula. A signal $\mathbf{u}: [0, T] \rightarrow \mathbb{R}^{|\text{Var}|}$ is a *falsifying input* if $\llbracket \mathcal{M}(\mathbf{u}), \varphi \rrbracket < 0$; the latter implies $\mathcal{M}(\mathbf{u}) \not\models \varphi$.

The use of quantitative robust semantics $\llbracket \mathcal{M}(\mathbf{u}), \varphi \rrbracket \in \mathbb{R} \cup \{\infty, -\infty\}$ in the above problem enables the use of hill-climbing optimization.

Definition 6 (hill climbing-guided falsification). Assume the setting in Definition 5. For finding a falsifying input, the methodology of *hill climbing-guided falsification* is presented in Algorithm 1.

Here the function HILL-CLIMB makes a guess of an input signal \mathbf{u}_k , aiming at minimizing the robustness $\llbracket \mathcal{M}(\mathbf{u}_k), \varphi \rrbracket$. It does so, learning from the previous observations $(\mathbf{u}_l, \llbracket \mathcal{M}(\mathbf{u}_l), \varphi \rrbracket)_{l \in [1, k-1]}$ of input signals $\mathbf{u}_1, \dots, \mathbf{u}_{k-1}$ and their corresponding robustness values (cf. Table 1).

The HILL-CLIMB function can be implemented by various stochastic optimization algorithms. Examples are CMA-ES [6] (used in our experiments), SA, and GNM [30].

3 Our Multi-armed Bandit-Based Falsification Algorithm

In this section, we present our contribution, namely a falsification algorithm that addresses the scale problem in Boolean superposition (see Sect. 1). The main novelties in the algorithm are as follows.

1. **(Use of MAB algorithms)** For binary Boolean connectives, unlike most works in the field, we do not superpose the robustness values of the constituent formulas φ_1 and φ_2 using a fixed operator (such as \sqcap and \sqcup in (2)). Instead, we view the situation as an instance of the multi-armed bandit problem (MAB): we use an algorithm for MAB to choose one formula φ_i to focus on (here $i \in \{1, 2\}$); and then we apply hill climbing-guided falsification to the chosen formula φ_i .
2. **(Hill-climbing gain as rewards in MAB)** For our integration of MAB and hill-climbing optimization, the technical challenge is find a suitable notion of reward for MAB. We introduce a novel notion that we call *hill-climbing gain*: it formulates the (downward) robustness gain that we would obtain by applying hill-climbing optimization, suitably normalized using the scale of previous robustness values.

Later, in Sect. 4, we demonstrate that combining those two features gives rise to falsification algorithms that successfully cope with the scale problem in Boolean superposition.

Our algorithms focus on a fragment of STL as target specifications. They are called (*disjunctive and conjunctive*) *safety properties*. In Sect. 3.1 we describe this fragment of STL, and introduce necessary adaptation of the semantics. After reviewing the MAB problem in Sect. 3.2, we present our algorithms in Sects. 3.3, 3.4.

Algorithm 1. Hill climbing-guided falsification

Require: a system model \mathcal{M} , an STL formula φ , and a budget K

```

1: function HILL-CLIMB-FALSIFY( $\mathcal{M}, \varphi, K$ )
2:    $rb \leftarrow \infty$ ;  $k \leftarrow 0$             $\triangleright rb$  is the smallest robustness so far, initialized to  $\infty$ 
3:   while  $rb \geq 0$  and  $k \leq K$  do
4:      $k \leftarrow k + 1$ 
5:      $\mathbf{u}_k \leftarrow \text{HILL-CLIMB}\left(\left(\mathbf{u}_l, \llbracket \mathcal{M}(\mathbf{u}_l), \varphi \rrbracket\right)_{l \in [1, k-1]}\right)$ 
6:      $rb_k \leftarrow \llbracket \mathcal{M}(\mathbf{u}_k), \varphi \rrbracket$ 
7:     if  $rb_k < rb$  then  $rb \leftarrow rb_k$ 
8:    $\mathbf{u} \leftarrow \begin{cases} \mathbf{u}_k & \text{if } rb < 0, \text{ that is, } rb_k = \llbracket \mathcal{M}(\mathbf{u}_k), \varphi \rrbracket < 0 \\ \text{Failure} & \text{otherwise, that is, no falsifying input found within budget } K \end{cases}$ 
9:   return  $\mathbf{u}$ 

```

3.1 Conjunctive and Disjunctive Safety Properties

Definition 7 (conjunctive/disjunctive safety property). An STL formula of the form $\square_I(\varphi_1 \wedge \varphi_2)$ is called a *conjunctive safety property*; an STL formula of the form $\square_I(\varphi_1 \vee \varphi_2)$ is called a *disjunctive safety property*.

It is known that, in industry practice, a majority of specifications is of the form $\square_I(\varphi_1 \rightarrow \varphi_2)$, where φ_1 describes a trigger and φ_2 describes a countermeasure that should follow. This property is equivalent to $\square_I(\neg\varphi_1 \vee \varphi_2)$, and is therefore a disjunctive safety property.

In Sects. 3.3, 3.4, we present two falsification algorithms, for conjunctive and disjunctive safety properties respectively. For the reason we just discussed, we expect the disjunctive algorithm should be more important in real-world application scenarios. In fact, the disjunctive algorithm turns out to be more complicated, and it is best introduced as an extension of the conjunctive algorithm.

We define the restriction of robust semantics to a (sub)set of time instants. Note that we do not require $\mathcal{S} \subseteq [0, T]$ to be a single interval.

Definition 8 ($\llbracket \mathbf{w}, \psi \rrbracket_{\mathcal{S}}$, robustness restricted to $\mathcal{S} \subseteq [0, T]$). Let $\mathbf{w}: [0, T] \rightarrow \mathbb{R}^{|Var|}$ be a signal, ψ be an STL formula, and $\mathcal{S} \subseteq [0, T]$ be a subset. We define the *robustness* of \mathbf{w} under ψ *restricted to* \mathcal{S} by

$$\llbracket \mathbf{w}, \psi \rrbracket_{\mathcal{S}} := \sqcap_{t \in \mathcal{S}} \llbracket \mathbf{w}^t, \psi \rrbracket. \quad (3)$$

Obviously, $\llbracket \mathbf{w}, \psi \rrbracket_{\mathcal{S}} < 0$ implies that there exists $t \in \mathcal{S}$ such that $\llbracket \mathbf{w}^t, \psi \rrbracket < 0$. We derive the following easy lemma; it is used later in our algorithm.

Lemma 9. *In the setting of Definition 8, consider a disjunctive safety property $\varphi \equiv \square_I(\varphi_1 \vee \varphi_2)$, and let $\mathcal{S} := \{t \in I \cap [0, T] \mid \llbracket \mathbf{w}^t, \varphi_1 \rrbracket < 0\}$. Then $\llbracket \mathbf{w}, \varphi_2 \rrbracket_{\mathcal{S}} < 0$ implies $\llbracket \mathbf{w}, \square_I(\varphi_1 \vee \varphi_2) \rrbracket < 0$.* \square

3.2 The Multi-Armed Bandit (MAB) Problem

The *multi-armed bandit* (MAB) problem describes a situation where,

- a gambler sits in front of a row A_1, \dots, A_n of slot machines;
- each slot machine A_i gives, when its arm is played (i.e. in each attempt), a reward according to a prescribed (but unknown) probability distribution μ_i ;
- and the goal is to maximize the cumulative reward after a number of attempts, playing a suitable arm in each attempt.

The best strategy of course is to keep playing the best arm A_{\max} , i.e. the one whose average reward $\text{avg}(\mu_{\max})$ is the greatest. This best strategy is infeasible, however, since the distributions μ_1, \dots, μ_n are initially unknown. Therefore the gambler must learn about μ_1, \dots, μ_n through attempts.

The MAB problem exemplifies the “learning by trying” paradigm of *reinforcement learning*, and is thus heavily studied. The greatest challenge is to balance between *exploration* and *exploitation*. A greedy (i.e. exploitation-only) strategy will play the

arm whose empirical average reward is the maximum. However, since the rewards are random, this way the gambler can miss another arm whose real performance is even better but which is yet to be found so. Therefore one needs to mix exploration, too, occasionally trying empirically non-optimal arms, in order to identify their true performance.

The relevance of MAB to our current problem is as follows. Falsifying a conjunctive safety property $\square_I(\varphi_1 \wedge \varphi_2)$ amounts to finding a time instant $t \in I$ at which either φ_1 or φ_2 is falsified. We can see the two subformulas (φ_1 and φ_2) as two arms, and this constitutes an instance of the MAB problem. In particular, playing an arm translates to a falsification attempt by hill climbing, and collecting rewards translates to spending time to minimize the robustness. We show in Sects. 3.3–3.4 that this basic idea extends to disjunctive safety properties $\square_I(\varphi_1 \vee \varphi_2)$, too.

Algorithm 2. The ε -greedy algorithm for multi-armed bandits

Require: the setting of Def. 10, and a constant $\varepsilon > 0$ (typically very small)

At the k -th attempt, choose the arm A_{i_k} as follows

- 1: $j_{\text{emp-opt}} \leftarrow \arg \max_{j \in [1, n]} R(j, k - 1)$ ▷ the arm that is empirically optimal
 - 2: Sample $i_k \in [1, n]$ from the distribution

$$\begin{cases} j_{\text{emp-opt}} \mapsto (1 - \varepsilon) + \frac{\varepsilon}{n} \\ j \mapsto \frac{\varepsilon}{n} & \text{for each } j \in [1, n] \setminus \{j_{\text{emp-opt}}\} \end{cases}$$
 - 3: **return** i_k
-

A rigorous formulation of the MAB problem is presented for the record.

Definition 10 (the multi-armed bandit problem). The *multi-armed bandit* (MAB) problem is formulated as follows.

Input: arms (A_1, \dots, A_n) , the associated probability distributions μ_1, \dots, μ_n over \mathbb{R} , and a time horizon $H \in \mathbb{N} \cup \{\infty\}$.

Goal: synthesize a sequence $A_{i_1} A_{i_2} \dots A_{i_H}$, so that the cumulative reward $\sum_{k=1}^H \text{rew}_k$ is maximized. Here the reward rew_k of the k -th attempt is sampled from the distribution μ_{i_k} associated with the arm A_{i_k} played at the k -th attempt.

We introduce some notations for later use. Let $(A_{i_1} \dots A_{i_k}, \text{rew}_1 \dots \text{rew}_k)$ be a *history*, i.e. the sequence of arms played so far (here $i_1, \dots, i_k \in [1, n]$), and the sequence of rewards obtained by those attempts (rew_l is sampled from μ_{i_l}).

For an arm A_j , its *visit count* $N(j, A_{i_1} A_{i_2} \dots A_{i_k}, \text{rew}_1 \text{rew}_2 \dots \text{rew}_k)$ is given by the number of occurrences of A_j in $A_{i_1} A_{i_2} \dots A_{i_k}$. Its *empirical average reward* $R(j, A_{i_1} A_{i_2} \dots A_{i_k}, \text{rew}_1 \text{rew}_2 \dots \text{rew}_k)$ is given by $\sum_{l \in \{l \in [1, k] | i_l = j\}} \text{rew}_l$, i.e. the average return of the arm A_j in the history. When the history is obvious from the context, we simply write $N(j, k)$ and $R(j, k)$.

MAB Algorithms. There have been a number of algorithms proposed for the MAB problem; each of them gives a *strategy* (also called a *policy*) that tells which arm to play, based on the previous attempts and their rewards. The focus here is how to resolve the exploration-exploitation trade-off. Here we review two well-known algorithms.

The ε -Greedy Algorithm. This is a simple algorithm that spares a small fraction ε of chances for empirically non-optimal arms. The spared probability ε is uniformly distributed. See Algorithm 2.

The UCB1 Algorithm. The UCB1 (*upper confidence bound*) algorithm is more complex; it comes with a theoretical upper bound for *regrets*, i.e. the gap between the expected cumulative reward and the optimal (but infeasible) cumulative reward (i.e. the result of keep playing the optimal arm A_{\max}). It is known that the UCB1 algorithm's regret is at most $O(\sqrt{nH \log H})$ after H attempts, improving the naive random strategy (which has the expected regret $O(H)$).

See Algorithm 3. The algorithm is deterministic, and picks the arm that maximizes the value shown in Line 1. The first term $R(j, k - 1)$ is the *exploitation* factor, reflecting the arm's empirical performance. The second term is the *exploration* factor. Note that it is bigger if the arm A_j has been played less frequently. Note also that the exploration factor eventually decays over time: the denominator grows roughly with $O(k)$, while the numerator grows with $O(\ln k)$.

Algorithm 3. The UCB1 algorithm for multi-armed bandits

Require: the setting of Def. 10, and a constant $c > 0$

At the k -th attempt, choose the arm A_{i_k} as follows

$$1: \quad i_k \leftarrow \arg \max_{j \in [1, n]} \left(R(j, k - 1) + c \sqrt{\frac{2 \ln(k - 1)}{N(j, k - 1)}} \right)$$

2: **return** i_k

Algorithm 4. Our MAB-guided algorithm I: *conjunctive* safety properties

Require: a system model \mathcal{M} , an STL formula $\varphi \equiv \square_I(\varphi_1 \wedge \varphi_2)$, and a budget K

1: **function** MAB-FALSIFY-CONJ-SAFETY(\mathcal{M}, φ, K)

2: $\text{rb} \leftarrow \infty$; $k \leftarrow 0$

▷ rb is the smallest robustness seen so far, for either $\square_I \varphi_1$ or $\square_I \varphi_2$

3: **while** $rb > 0$ and $k < K$ **do**

▷ iterate if not yet falsified, and within budget

$$4: \quad k \leftarrow \overline{k} + 1$$

$$i_k \leftarrow \text{MAB}\left(\left(\varphi_1, \varphi_2\right), \left(\mathcal{R}(\varphi_1), \mathcal{R}(\varphi_2)\right), \varphi_{i_1} \dots \varphi_{i_{k-1}}, \text{rew}_1 \dots \text{rew}_{k-1}\right)$$

▷ an MAB choice of $i_k \in \{1, 2\}$ for optimizing the reward $\mathcal{R}(\varphi_{i_k})$

6: $\mathbf{u}_k \leftarrow \text{HILL-CLIMB} \left((\mathbf{u}_l, \mathbf{rb}_l) \right)_{l \in [1, k-1] \text{ such that } i_l = i_k}$

- ▷ suggestion of the next input \mathbf{u}_k by hill climbing, based on the previous observations on the formula φ_{i_k} (those on the other formula are ignored)

$$7: \quad \quad \quad \mathbf{rb}_k \leftarrow [\![\mathcal{M}(\mathbf{u}_k), \square_I \varphi_{i_k}]\!]$$

if $rb_k < rb$ **then** $rb \leftarrow rb_k$

$$9: \quad \mathbf{u} \leftarrow \begin{cases} \mathbf{u}_k & \text{if } \mathbf{r}\mathbf{b} < 0 \end{cases}$$

\mathbf{u}^* | Failure otherwise, that is, no falsifying input found within budget K
return \mathbf{u}^*

10. Return a

Algorithm 5. Our MAB-guided algorithm II: *disjunctive* safety properties

Require: a system model \mathcal{M} , an STL formula $\varphi \equiv \square_I(\varphi_1 \vee \varphi_2)$, and a budget K

1: **function** MAB-FALSIFY-DISJ-SAFETY(\mathcal{M}, φ, K)

The same as Algorithm 4, except that Line 7 is replaced by the following Line 7'.

7': $\text{rb}_k \leftarrow \llbracket \mathcal{M}(\mathbf{u}_k), \varphi_{i_k} \rrbracket_{\mathcal{S}_k}$ where $\mathcal{S}_k = \{t \in I \cap [0, T] \mid \llbracket \mathcal{M}(\mathbf{u}_k^t), \varphi_{i_k} \rrbracket < 0\}$

▷ here $\varphi_{i_k}^-$ denotes the other formula than φ_{i_k} , among φ_1, φ_2

3.3 Our MAB-Guided Algorithm I: Conjunctive Safety Properties

Our first algorithm targets at conjunctive safety properties. It is based on our identification of MAB in a Boolean conjunction in falsification—this is as we discussed just above Definition 10. The technical novelty lies in the way we combine MAB algorithms and hill-climbing optimization; specifically, we introduce the notion of *hill-climbing gain* as a reward notion in MAB (Definition 11). This first algorithm paves the way to the one for disjunctive safety properties, too (Sect. 3.4).

The algorithm is in Algorithm 4. Some remarks are in order.

Algorithm 4 aims to falsify a conjunctive safety property $\varphi \equiv \square_I(\varphi_1 \wedge \varphi_2)$. Its overall structure is to *interleave* two sequences of falsification attempts, both of which are hill climbing-guided. These two sequences of attempts aim to falsify $\square_I\varphi_1$ and $\square_I\varphi_2$, respectively. Note that $\llbracket \mathcal{M}(\mathbf{u}), \varphi \rrbracket \leq \llbracket \mathcal{M}(\mathbf{u}), \square_I\varphi_1 \rrbracket$, therefore falsification of $\square_I\varphi_1$ implies falsification of φ ; the same holds for $\square_I\varphi_2$, too.

In Line 5 we run an MAB algorithm to decide which of $\square_I\varphi_1$ and $\square_I\varphi_2$ to target at in the k -th attempt. The function MAB takes the following as its arguments: (1) the list of arms, given by the formulas φ_1, φ_2 ; (2) their rewards $\mathcal{R}(\varphi_1), \mathcal{R}(\varphi_2)$; (3) the history $\varphi_{i_1} \dots \varphi_{i_{k-1}}$ of previously played arms ($i_l \in \{1, 2\}$); and 4) the history $\text{rew}_1 \dots \text{rew}_{k-1}$ of previously observed rewards. This way, the type of the MAB function in Line 5 matches the format in Definition 10, and thus the function can be instantiated with any MAB algorithm such as Algorithms 2–3.

The only missing piece is the definition of the rewards $\mathcal{R}(\varphi_1), \mathcal{R}(\varphi_2)$. We introduce the following notion, tailored for combining MAB and hill climbing.

Definition 11 (hill-climbing gain). In Algorithm 4, in Line 5, the reward $\mathcal{R}(\varphi_i)$ of the arm φ_i (where $i \in \{1, 2\}$) is defined by

$$\mathcal{R}(\varphi_i) = \begin{cases} \frac{\max\text{-rb}(i, k-1) - \text{last}\text{-rb}(i, k-1)}{\max\text{-rb}(i, k-1)} & \text{if } \varphi_i \text{ has been played before} \\ 0 & \text{otherwise} \end{cases}$$

Here $\max\text{-rb}(i, k-1) := \max\{\text{rb}_l \mid l \in [1, k-1], i_l = i\}$ (i.e. the greatest rb_l so far, in those attempts where φ_i was played), and $\text{last}\text{-rb}(i, k-1) := \text{rb}_{l_{\text{last}}}$ with l_{last} being the greatest $l \in [1, k-1]$ such that $i_l = i$ (i.e. the last rb_l for φ_i).

Since we try to minimize the robustness values rb_l through falsification attempts, we can expect that rb_l for a fixed arm φ_i decreases over time. (In the case of the hill-climbing algorithm CMA-ES that we use, this is in fact guaranteed). Therefore the value

$\max\text{-rb}(i, k - 1)$ in the definition of $\mathcal{R}(\varphi_i)$ is the first observed robustness value. The numerator $\max\text{-rb}(i, k - 1) - \text{last-rb}(i, k - 1)$ then represents how much robustness we have reduced so far by hill climbing—hence the name “hill-climbing gain.” The denominator $\max\text{-rb}(i, k - 1)$ is there for normalization.

In Algorithm 4, the value rb_k is given by the robustness $\llbracket \mathcal{M}(\mathbf{u}_k), \square_I \varphi_{i_k} \rrbracket$. Therefore the MAB choice in Line 5 essentially picks i_k for which hill climbing yields greater effect (but also taking exploration into account—see Sect. 3.2).

In Line 6 we conduct hill-climbing optimization—see Sect. 2.2. The function HILL-CLIMB learns from the previous attempts $\mathbf{u}_{l_1}, \dots, \mathbf{u}_{l_m}$ regarding the same formula φ_{i_k} , and their resulting robustness values $\text{rb}_{l_1}, \dots, \text{rb}_{l_m}$. Then it suggests the next input signal \mathbf{u}_k that is likely to minimize the (unknown) function that underlies the correspondences $[\mathbf{u}_{l_j} \mapsto \text{rb}_{l_j}]_{j \in [1, m]}$.

Lines 6–8 read as follows: the hill-climbing algorithm suggests a single input \mathbf{u}_k , which is then selected or rejected (Line 8) based on the robustness value it yields (Line 7). We note that this is a simplified picture: in our implementation that uses CMA-ES (it is an evolutionary algorithm), we maintain a population of some ten particles, and each of them is moved multiple times (our choice is three times) before the best one is chosen as \mathbf{u}_k .

3.4 Our MAB-Guided Algorithm II: Disjunctive Safety Properties

The other main algorithm of ours aims to falsify a *disjunctive* safety property $\varphi \equiv \square_I(\varphi_1 \vee \varphi_2)$. We believe this problem setting is even more important than the conjunctive case, since it encompasses conditional safety properties (i.e. of the form $\square_I(\varphi_1 \rightarrow \varphi_2)$). See Sect. 3.1 for discussions.

In the disjunctive setting, the challenge is that falsification of $\square_I \varphi_i$ (with $i \in \{1, 2\}$) does *not* necessarily imply falsification of $\square_I(\varphi_1 \vee \varphi_2)$. This is unlike the conjunctive setting. Therefore we need some adaptation of Algorithm 4, so that the two interleaved sequences of falsification attempts for φ_1 and φ_2 are not totally independent of each other. Our solution consists of *restricting* time instants to those where φ_2 is false, in a falsification attempt for φ_1 (and vice versa), in the way described in Definition 8.

Algorithm 5 shows our MAB-guided algorithm for falsifying a disjunctive safety property $\square_I(\varphi_1 \vee \varphi_2)$. The only visible difference is that Line 7 in Algorithm 4 is replaced with Line 7’. The new Line 7’ measures the quality of the suggested input signal \mathbf{u}_k in the way restricted to the region \mathcal{S}_k in which the other formula is already falsified. Lemma 9 guarantees that, if $\text{rb}_k < 0$, then indeed the input signal \mathbf{u}_k falsifies the original specification $\square_I(\varphi_1 \vee \varphi_2)$.

The assumption that makes Algorithm 5 sensible is that, although it can be hard to find a time instant at which both φ_1 and φ_2 are false (this is required in falsifying $\square_I(\varphi_1 \vee \varphi_2)$), falsifying φ_1 (or φ_2) individually is not hard. Without this assumption, the region \mathcal{S}_k in Line 7’ would be empty most of the time. Our experiments in Sect. 4 demonstrate that this assumption is valid in many problem instances, and that Algorithm 5 is effective.

4 Experimental Evaluation

We name MAB-UCB and MAB- ϵ -greedy the two versions of MAB algorithm using strategies ϵ -Greedy (see Algorithm 2) and UCB1 (see Algorithm 3). We compared the proposed approach (both versions MAB-UCB and MAB- ϵ -greedy) with a state-of-the-art falsification framework, namely Breach [11]. Breach encapsulates several hill-climbing optimization algorithms, including *CMA-ES* (*covariance matrix adaptation evolution strategy*) [6], *SA* (*simulated annealing*), *GNM* (*global Nelder-Mead*) [30], etc. According to our experience, CMA-ES outperforms other hill-climbing solvers in Breach, so the experiments for both Breach and our approach rely on the CMA-ES solver.

Experiments have been executed using Breach 1.2.13 on an Amazon EC2 c4.large instance, 2.9 GHz Intel Xeon E5-2666, 2 virtual CPU cores, 4 GB RAM.

Benchmarks. We selected three benchmark models from the literature, each one having different specifications. The first one is the *Automatic Transmission* (AT) model [16, 24]. It has two input signals, $throttle \in [0, 100]$ and $brake \in [0, 325]$, and computes the car's *speed*, engine rotation in rounds per minute *rpm*, and the automatically selected *gear*. The specifications concern the relation between the three output signals to check whether the car is subject to some unexpected or unsafe behaviors. The second benchmark is the *Abstract Fuel Control* (AFC) model [16, 25]. It takes two input signals, $pedal\ angle \in [8.8, 90]$ and $engine\ speed \in [900, 1100]$, and outputs the critical signal *air-fuel ratio* (*AF*), which influences fuel efficiency and car performance. The value is expected to be close to a reference value *AFref*; $mu \equiv |AF - AFref| / AFref$ is the deviation of *AF* from *AFref*. The specifications check whether this property holds under both *normal mode* and *power enrichment mode*. The third benchmark is a model of a *magnetic levitation system with a NARMA-L2 neurocontroller* (NN) [7, 16]. It takes one input signal, $Ref \in [1, 3]$, which is the reference for the output signal *Pos*, the position of a magnet suspended above an electromagnet. The specifications say that the position should approach the reference signal in a few seconds when these two are not close.

Table 2. Benchmark sets Bbench and Sbench

(a) Bbench (here $\delta_{t'}(\mathbf{w})$ represents $\mathbf{w}^t(t') - \mathbf{w}^t(0)$).			(b) Sbench		
Bench	Spec ID	Formula	Specification	Parameter	Spec ID
AT	AT1	$\square_{[0,30]}((gear = 3) \rightarrow (speed > \rho))$		$\rho \in \{20.6, 20.4, 20.2, 20, 19.8\}$	AT1 ¹
	AT2	$\square_{[0,30]}((gear = 4) \rightarrow (speed > \rho))$		$\rho \in \{43, 41, 39, 37, 35\}$	AT1 ²
	AT3	$\square_{[0,30]}((gear = 4) \rightarrow (rpm > \rho))$		$\rho \in \{700, 800, 900, 1000, 1100\}$	AT1 ³
	AT4	$\square_{[0,30-\tau]}((\delta_{10}(rpm) > 2000) \rightarrow (\delta_r(gear) > 0))$	$\tau \in \{15, 16, 17, 18, 19\}$		AT1 ⁴
	AT5	$\square_{[0,30]}((speed < \rho) \wedge (RPM < 4780))$		$\rho \in \{130, 131, 132, 133, 134, 135, 136, 137\}$	AT1 ⁵
	AT6	$\square_{[0,26]}((\delta_4(speed) > \rho) \rightarrow (\delta_4(gear) > 0))$		$\rho \in \{20, 25, 30, 35, 40\}$	AT5 ⁴
	AT7	$\square_{[0,30-\tau]}((\delta_r(speed) > 30) \rightarrow (\delta_r(gear) > 0))$	$\tau \in \{2, 3, 4, 5, 6, 7, 8\}$		AT5 ⁵
AFC	AFC1	$\square_{[11..50]}((controller_mode = 0) \rightarrow (mu < \rho))$		$\rho \in \{0.16, 0.17, 0.18, 0.19, 0.2\}$	AT5 ⁶
	AFC2	$\square_{[11..50]}((controller_mode = 1) \rightarrow (mu < \rho))$		$\rho \in \{0.222, 0.224, 0.226, 0.228, 0.23\}$	AT5 ⁷
NN	$close \equiv Pos - Ref \leq \rho + \alpha * Ref $ $reach \equiv \Diamond_{[0,2]}(\square_{[0,1]}(close))$				AT5 ⁸
	NN1	$\square_{[0,18]}(\neg close \rightarrow reach), \alpha = 0.04$		$\rho \in \{0.001, 0.002, 0.003, 0.004, 0.005\}$	AFC1 ₁
	NN1	$\square_{[0,18]}(\neg close \rightarrow reach), \alpha = 0.03$		$\rho \in \{0.001, 0.002, 0.003, 0.004, 0.005\}$	AFC1 ₂
					AFC1 ₃
					$mu \quad k \in \{0,1,2,3\}$
					AFC1 ₄
					AFC1 ₅

We built the benchmark set Bbench , as shown in Table 2a that reports the name of the model and its specifications (ID and formula). In total, we found 11 specifications. In order to increase the benchmark set and obtain specifications of different complexity, we artificially modified a constant (turned into a parameter named τ if it is contained in a time interval, named ρ otherwise) of the specification: for each specification S , we generated m different versions, named as S_i with $i \in \{1, \dots, m\}$; the complexity of the specification (in terms of difficulty to falsify it) increases with increasing i .² In total, we produced 60 specifications. Column *parameter* in the table shows which concrete values we used for the parameters ρ and τ . Note that all the specifications but one are disjunctive safety properties (i.e., $\square_I(\varphi_1 \vee \varphi_2)$), as they are the most difficult case and they are the main target of our approach; we just add AT5 as example of conjunctive safety property (i.e., $\square_I(\varphi_1 \wedge \varphi_2)$).

Our approach has been proposed with the aim of tackling the scale problem. Therefore, to better show how our approach mitigates this problem, we generated a second benchmark set Sbench as follows. We selected 15 specifications from Bbench (with concrete values for the parameters) and, for each specification S , we changed the corresponding Simulink model by multiplying one of its outputs by a factor 10^k , with $k \in \{-2, 0, 1, 2, 3\}$ (note that we also include the original one using scale factor 10^0); the specification has been modified accordingly, by multiplying with the scale factor the constants that are compared with the scaled output. We name a specification S scaled with factor 10^k as S^k . Table 2b reports the IDs of the original specifications, the output that has been scaled, and the used scaled factors; in total, the benchmark set Sbench contains 60 specifications.

Experiment. In our context, an *experiment* consists in the execution of an approach A (either *Breach*, MAB- ϵ -greedy, or MAB-UCB) over a specification S for 30 *trials*, using different initial seeds. For each experiment, we record the *success SR* as the number of trials in which a falsifying input was found, and average execution *time* of the trials. Complete experimental results are reported in Appendix A in the extended version [37]³. We report aggregated results in Table 3.

For benchmark set Bbench , it reports aggregated results for each group of specifications obtained from S (i.e., all the different versions S_i obtained by changing the value of the parameter); for benchmark set Sbench , instead, results are aggregated for each scaled specification S^k (considering the versions S_i^k obtained by changing the parameter value). We report minimum, maximum and average number of successes SR, and time in seconds. For MAB- ϵ -greedy and MAB-UCB, both for SR and time, we also report the average percentage difference⁴ (Δ) w.r.t. to the corresponding value of *Breach*.

Comparison. In the following, we compare two approaches $A_1, A_2 \in \{\text{Breach}, \text{MAB}-\epsilon\text{-greedy}, \text{MAB}-\text{UCB}\}$ by comparing the number of their successes SR and average execution *time* using the non-parametric Wilcoxon signed-rank test with 5%

² Note that we performed this classification based on the falsification results of *Breach*.

³ The code, models, and specifications are available online at <https://github.com/ERATOMMSD/FalStar-MAB>.

⁴ $\Delta = ((m - b) * 100) / (0.5 * (m + b))$ where m is the result of MAB and b the one of *Breach*.

Table 3. Aggregated results for benchmark sets Bbench and Sbench (SR: # successes out 30 trials. Time in secs. Δ : percentage difference w.r.t. Breach). Outperformance cases are highlighted, indicated by positive Δ of SR, and negative Δ of time.

Spec. ID	Breach						MAB- ϵ -greedy						MAB-UCB									
	SR (/30)			time (sec.)			SR (/30)			time (sec.)			SR (/30)			time (sec.)						
	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg	Δ			
AT1	14	25	20.2	125	361.2	223.1	24	30	28.6	35.7	62.7	213.4	106.4	-73.4	28	30	29.2	37.8	45.1	146.8	77.4	-97.1
AT2	11	30	20.2	14	390.6	209.8	30	30	30	43.9	11.9	126.3	54.5	-96.9	27	30	29.4	42.2	17.7	92.5	36.8	-112.1
AT3	29	30	29.4	2.3	22.2	14.2	30	30	30	2	2.5	7	3.5	-82.9	30	30	30	2	2.5	3.6	3	-88.6
AT4	18	30	25.8	19.5	265.3	109.6	29	30	29.8	16	7.8	45.1	24.4	-105	30	30	30	16.6	6.2	36.2	22.2	-113.5
AT5	6	23	14.1	203.1	525.9	366.2	26	30	28.5	72.1	35.2	149	93.7	-120.6	26	30	28.2	71.4	37.7	154.1	99.2	-116.8
AT6	5	29	22.8	30.1	509.5	157	21	30	27	28	2.3	300	95.1	-98.3	22	30	27	27.7	2.9	247.3	86.1	-99.4
AT7	15	30	26.6	12.2	314	81.5	20	30	28.6	8.4	29	28.9	49.9	-92	23	30	29	10.3	5.5	223.3	42.9	-88.3
AFC1	6	30	14.4	124.8	565.6	413.5	4	28	12	-28.4	171	568.4	446	-10.8	5	30	16.4	9.7	98.7	559.8	389.9	-9.3
AFC2	2	30	18	80.7	582.3	343.4	5	30	20	23.8	43.2	547.8	301.9	-23.8	5	30	20	22.9	59.4	568.4	320.5	-11.1
NN1	17	25	20.8	212.9	384.7	292.9	14	27	20.2	-4.5	189.5	422.8	320.3	6.2	17	28	22.6	7.3	148.2	403.3	272.3	-11.8
NN2	27	28	27.2	55.5	93.4	73.1	30	30	30	9.8	11	39.3	26.3	-97.8	30	30	9.8	14.6	38.2	27.4	-92.3	
AT1 ⁻²	30	30	30	42.5	97.4	56.9	28	30	29	-3.4	75.6	178.3	118.7	68.7	28	30	29.4	-2.1	54.3	136.3	80.3	33.3
AT1 ⁰	14	25	20.2	125	361.2	223.1	24	30	28.6	35.7	62.7	213.4	106.4	-73.4	28	30	29.2	37.8	45.1	146.8	77.4	-97.1
AT1 ¹	4	21	15.4	204.5	527.6	310.2	25	30	29	68.4	49	234.7	102.1	-108	27	29	28.2	64.5	77.5	128.7	105.1	-93
AT1 ³	8	24	19.8	164	471.7	240.1	29	30	29.8	44.6	67.5	170.6	101.9	-77.3	29	30	29.4	43.4	55.4	104.8	80.6	-93.6
AT5 ⁻²	29	30	29.6	61.1	163.7	102	25	30	27.8	-6.4	76.9	139.5	111.9	12.6	28	30	29.4	-0.7	48.5	131.9	85.7	-17
AT5 ⁰	6	18	11.2	291.1	525.9	423.1	28	30	28.4	90.5	80.2	151.3	107.4	-117.7	26	30	28	89.4	68.3	154.1	114.9	-114.5
AT5 ¹	0	2	0.4	566.4	600	593.3	27	30	28.4	194.8	70.7	184.5	110.3	-138.5	25	30	27.6	194.1	83.1	150	123.7	-131.2
AT5 ³	0	1	0.2	586.4	600	597.3	27	30	28.6	197.2	66.8	163.3	102.5	-142.3	27	29	28	197.2	80.4	160.9	111.9	-137.4
AFC1 ⁰	6	30	14.4	124.8	565.6	413.5	4	29	16.4	8.5	115.1	559.9	411.1	-2.8	5	30	16.4	9.7	98.7	559.8	389.9	-9.3
AFC1 ¹	7	30	16.6	99	548.2	393.3	3	29	10.8	-60.9	198.1	587.6	465.8	24.6	7	29	17.8	10.3	105.7	527.3	354.3	-10.3
AFC1 ²	0	12	5.2	434.4	600	535.8	3	28	11.6	96.2	180.8	577.6	463	-20.7	4	30	17	127	73.7	556.3	374.5	-47.3
AFC1 ³	1	12	4.8	425.7	587.4	532.6	3	30	14.4	109	138	585.5	436.5	-28	7	30	15	113	77.1	553.4	403.7	-39.9

level of significance⁵ [35]; the null hypothesis is that there is no difference in applying A_1 A_2 in terms of the compared measure (SR or time).

4.1 Evaluation

We evaluate the proposed approach with some research questions.

RQ1 Which is the best MAB algorithm for our purpose?

In Sect. 3.2, we described that the proposed approach can be executed using two different strategies for choosing the arm in the MAB problem, namely MAB- ϵ -greedy and MAB-UCB. We here assess which one is better in terms of SR and time. From the results in Table 3, it seems that MAB-UCB provides slightly better performance in terms of SR; this has been confirmed by the Wilcoxon test applied over all the experiments (i.e., on the non-aggregated data reported in Appendix A in the extended version [37]): the null hypothesis that using anyone of the two strategies has no impact on SR is rejected with p -value equal to 0.005089, and the alternative hypothesis that SR is better is accepted with p -value = 0.9975; in a similar way, the null hypothesis that there is no difference in terms of time is rejected with p -value equal to 3.495e–06, and the alternative hypothesis that is MAB-UCB is faster is accepted with p -value = 1. Therefore, in the following RQs, we compare Breach with only the MAB-UCB version of our approach.

⁵ We checked that the distributions are not normal with the non-parametric Shapiro-Wilk test.

RQ2 Does the proposed approach effectively solve the scale problem?

We here assess if our approach is effective in tackling the scale problem. Table 4 reports the complete experimental results over Sbench for Breach and MAB-UCB; for each specification S , all its scaled versions are reported in increasing order of the scaling factor. We observe that changing the scaling factor affects (sometimes greatly) the number of successes SR of Breach; for example, for AT5₅ and AT5₇ it goes from 30 to 0. For MAB-UCB, instead, SR is similar across the scaled versions of each specification: this shows that the approach is robust w.r.t. to the scale problem as the “hill-climbing gain” reward in Definition 11 eliminates the impact of scaling and UCB1 algorithm balances the exploration and exploitation of two sub-formulas. The observation is confirmed by the Wilcoxon test over SR: the null hypothesis is rejected with p -value = 1.808e–09, and the alternative hypothesis accepted with p -value = 1. Instead, the null hypothesis that there is no difference in terms of time cannot be rejected with p -value = 0.3294.

RQ3 How does the proposed process behave with not scaled benchmarks?

In RQ2, we checked whether the proposed approach is able to tackle the scale problem for which it has been designed. Here, instead, we are interested in investigating how it behaves on specifications that have not been artificially scaled (i.e., those in Bbench). From Table 3 (upper part), we observe that MAB-UCB is always better than Breach both in terms of SR and time, which is shown by the highlighted cases. This is confirmed by Wilcoxon test over SR and time: null hypotheses are rejected with p -values equal to, respectively, 6.02e–08 and 1.41e–08, and the alternative hypotheses that MAB-UCB is better are both accepted

Table 4. Experimental results – Sbench (SR: # successes out of 30 trials. Time in secs)

Spec. ID	Breach SR (/30)	MAB-UCB time (sec.)	Spec. ID	Breach SR (/30)	MAB-UCB time (sec.)	Spec. ID	Breach SR (/30)	MAB-UCB time (sec.)
AT1 ₁ ⁻²	30	51.3	AT5 ₄ ⁻²	30	61.1	30	48.5	AFC1 ₀ ⁰
AT1 ₁ ⁰	25	125	AT5 ₄ ⁰	18	291.1	28	94.5	AFC1 ₁ ¹
AT1 ₁ ¹	20	221.1	AT5 ₄ ¹	2	566.4	25	150	AFC1 ₂ ²
AT1 ₁ ³	23	170	AT5 ₄ ³	1	586.4	28	96.2	AFC1 ₃ ³
AT1 ₂ ⁻²	30	49	AT5 ₅ ⁻²	30	71.3	29	67.8	AFC1 ₀ ⁰
AT1 ₂ ⁰	22	187.5	AT5 ₅ ⁰	15	369.1	27	114	AFC1 ₁ ¹
AT1 ₂ ¹	21	204.5	AT5 ₅ ¹	0	600	29	83.1	AFC1 ₂ ²
AT1 ₂ ²	24	164	AT5 ₅ ²	0	600	27	113.8	AFC1 ₃ ³
AT1 ₃ ⁻²	30	42.5	AT5 ₆ ⁻²	29	110.2	28	103.3	AFC1 ₀ ⁰
AT1 ₃ ⁰	19	239.5	AT5 ₆ ⁰	10	438.2	30	68.3	AFC1 ₁ ¹
AT1 ₃ ¹	16	296.2	AT5 ₆ ¹	0	600	27	126.7	AFC1 ₂ ²
AT1 ₃ ³	21	209.8	AT5 ₆ ³	0	600	29	80.4	AFC1 ₃ ³
AT1 ₄ ⁻²	30	44.5	AT5 ₇ ⁻²	30	103.6	30	77.3	AFC1 ₀ ⁰
AT1 ₄ ⁰	21	202.2	AT5 ₇ ⁰	7	491.4	26	154.1	AFC1 ₁ ¹
AT1 ₄ ¹	16	301.7	AT5 ₇ ¹	0	600	27	134.3	AFC1 ₂ ²
AT1 ₄ ⁴	23	185.1	AT5 ₇ ⁴	0	600	29	108	AFC1 ₃ ³
AT1 ₅ ⁻²	30	97.4	AT5 ₈ ⁻²	29	163.7	30	131.9	AFC1 ₀ ⁰
AT1 ₅ ⁰	14	361.2	AT5 ₈ ⁰	6	525.9	29	143.6	AFC1 ₁ ¹
AT1 ₅ ¹	4	527.6	AT5 ₈ ¹	0	600	30	124.2	AFC1 ₂ ²
AT1 ₅ ⁵	8	471.7	AT5 ₈ ⁵	0	600	27	160.9	AFC1 ₃ ³
								AFC1 ₅ ⁵

with p -value = 1. This means that the proposed approach can also handle specifications that do not suffer from the scale problem, and so it can be used with any kind of specification.

RQ4 Is the proposed approach more effective than an approach based on rescaling?

A naïve solution to the scale problem could be to rescale the signals used in specification at the same scale. Thanks to the results of RQ2, we can compare to this possible baseline approach, using the scaled benchmark set Sbench. For example, AT5 suffers from the scale problem as *speed* is one order of magnitude less than *rpm*. However, from Table 3, we observe that the scaling that would be done by the baseline approach (i.e., running Breach over AT5¹) is not effective, as SR is 0.4/30, that is much lower than the original SR 14.1/30 of the unscaled approach using Breach. Our approach, instead, raises SR to 28.4/30 and to 27.6/30 using the two proposed versions. By monitoring Breach execution, we notice that the naïve approach fails because it tries to falsify $rpm < 4780$, which, however, is not falsifiable; our approach, instead, understands that it must try to falsify $speed < \rho$. More details are given in the extended version [37].

5 Conclusion and Future Work

In this paper, we propose a solution to the *scale problem* that affects falsification of specifications containing Boolean connectives. The approach combines multi-armed bandit algorithms with hill climbing-guided falsification. Experiments show that the approach is robust under the change of scales, and it outperforms a state-of-the-art falsification tool. The approach currently handles binary specifications. As future work, we plan to generalize it to complex specifications having more than two Boolean connectives.

References

- Adimoolam, A., Dang, T., Donzé, A., Kapinski, J., Jin, X.: Classification and coverage-based falsification for embedded control systems. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 483–503. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_24
- Akazaki, T., Hasuo, I.: Time robustness in MTL and expressivity in hybrid system falsification. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9207, pp. 356–374. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21668-3_21
- Akazaki, T., Kumazawa, Y., Hasuo, I.: Causality-aided falsification. In: Proceedings First Workshop on Formal Verification of Autonomous Vehicles, FVAV@iFM 2017. EPTCS, Turin, Italy, 19th September 2017, vol. 257, pp. 3–18 (2017)
- Akazaki, T., Liu, S., Yamagata, Y., Duan, Y., Hao, J.: Falsification of cyber-physical systems using deep reinforcement learning. In: Havelund, K., Peleska, J., Roscoe, B., de Vink, E. (eds.) FM 2018. LNCS, vol. 10951, pp. 456–465. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-95582-7_27
- Annpureddy, Y., Liu, C., Fainekos, G., Sankaranarayanan, S.: S-TALiRO: a tool for temporal logic falsification for hybrid systems. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 254–257. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19835-9_21

6. Auger, A., Hansen, N.: A restart CMA evolution strategy with increasing population size. In: Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2005, pp. 1769–1776. IEEE (2005)
7. Beale, M.H., Hagan, M.T., Demuth, H.B.: Neural Network Toolbox™ User’s Guide. The Mathworks Inc., Natick (1992)
8. Chen, X., Ábrahám, E., Sankaranarayanan, S.: Flow*: an analyzer for non-linear hybrid systems. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 258–263. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_18
9. Deshmukh, J., Jin, X., Kapinski, J., Maler, O.: Stochastic local search for falsification of hybrid systems. In: Finkbeiner, B., Pu, G., Zhang, L. (eds.) ATVA 2015. LNCS, vol. 9364, pp. 500–517. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24953-7_35
10. Dokhanchi, A., Yaghoubi, S., Hoxha, B., Fainekos, G.E.: Vacuity aware falsification for MTL request-response specifications. In: 13th IEEE Conference on Automation Science and Engineering, CASE 2017, Xi'an, China, 20–23 August 2017, pp. 1332–1337. IEEE (2017)
11. Donzé, A.: Breach, a toolbox for verification and parameter synthesis of hybrid systems. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 167–170. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_17
12. Donzé, A., Maler, O.: Robust satisfaction of temporal logic over real-valued signals. In: Chatterjee, K., Henzinger, T.A. (eds.) FORMATS 2010. LNCS, vol. 6246, pp. 92–106. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15297-9_9
13. Dreossi, T., Dang, T., Donzé, A., Kapinski, J., Jin, X., Deshmukh, J.V.: Efficient guiding strategies for testing of temporal properties of hybrid systems. In: Havelund, K., Holzmann, G., Joshi, R. (eds.) NFM 2015. LNCS, vol. 9058, pp. 127–142. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-17524-9_10
14. Dreossi, T., Dang, T., Piazza, C.: Parallelotope bundles for polynomial reachability. In: Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control, HSCC 2016, pp. 297–306. ACM, New York (2016)
15. Dreossi, T., Donzé, A., Seshia, S.A.: Compositional falsification of cyber-physical systems with machine learning components. In: Barrett, C., Davies, M., Kahsai, T. (eds.) NFM 2017. LNCS, vol. 10227, pp. 357–372. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-57288-8_26
16. Ernst, G., et al.: ARCH-COMP 2019 category report: Falsification. In: Frehse, G., Althoff, M. (eds.) 6th International Workshop on Applied Verification of Continuous and Hybrid Systems, ARCH19. EPiC Series in Computing, vol. 61 pp. 129–140. EasyChair (2019)
17. Fainekos, G.E., Pappas, G.J.: Robustness of temporal logic specifications for continuous-time signals. Theor. Comput. Sci. **410**(42), 4262–4291 (2009)
18. Fan, C., Qi, B., Mitra, S., Viswanathan, M., Duggirala, P.S.: Automatic reachability analysis for nonlinear hybrid models with C2E2. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9779, pp. 531–538. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41528-4_29
19. Ferrère, T., Nickovic, D., Donzé, A., Ito, H., Kapinski, J.: Interface-aware signal temporal logic. In: Ozay, N., Prabhakar, P. (eds.) Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control, HSCC 2019, Montreal, QC, Canada, 16–18 April 2019, pp. 57–66. ACM (2019)
20. Frehse, G., et al.: SpaceEx: scalable verification of hybrid systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 379–395. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_30
21. Fu, Z., Su, Z.: XSat: a fast floating-point satisfiability solver. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016, Part II. LNCS, vol. 9780, pp. 187–209. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_11

22. Gao, S., Avigad, J., Clarke, E.M.: δ -complete decision procedures for satisfiability over the reals. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS (LNAI), vol. 7364, pp. 286–300. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31365-3_23
23. Hasuo, I., Suenaga, K.: Exercises in nonstandard static analysis of hybrid systems. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 462–478. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31424-7_34
24. Hoxha, B., Abbas, H., Fainekos, G.E.: Benchmarks for temporal logic requirements for automotive systems. In: Frehse, G., Althoff, M. (eds.) 1st and 2nd International Workshop on Applied veRification for Continuous and Hybrid Systems, ARCH@CPSWeek 2014, Berlin, Germany, 14 April 2014/ARCH@CPSWeek 2015, Seattle, USA, 13 April 2015. EPiC Series in Computing, vol. 34, pp. 25–30. EasyChair (2014)
25. Jin, X., Deshmukh, J.V., Kapinski, J., Ueda, K., Butts, K.: Powertrain control verification benchmark. In: Proceedings of the 17th International Conference on Hybrid Systems: Computation and Control, HSCC 2014, pp. 253–262. ACM, New York (2014)
26. Kapinski, J., Deshmukh, J.V., Jin, X., Ito, H., Butts, K.: Simulation-based approaches for verification of embedded control systems: An overview of traditional and advanced modeling, testing, and verification techniques. IEEE Control. Syst. **36**(6), 45–64 (2016)
27. Kato, K., Ishikawa, F., Honiden, S.: Falsification of cyber-physical systems with reinforcement learning. In: 3rd Workshop on Monitoring and Testing of Cyber-Physical Systems, MT@CPSWeek 2018, Porto, Portugal, 10 April 2018, pp. 5–6. IEEE (2018)
28. Kuřátko, J., Ratschan, S.: Combined global and local search for the falsification of hybrid systems. In: Legay, A., Bozga, M. (eds.) FORMATS 2014. LNCS, vol. 8711, pp. 146–160. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10512-3_11
29. Liebrenz, T., Herber, P., Glesner, S.: Deductive verification of hybrid control systems modeled in simulink with KeYmaera X. In: Sun, J., Sun, M. (eds.) ICFEM 2018. LNCS, vol. 11232, pp. 89–105. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-02450-5_6
30. Luersen, M.A., Le Riche, R.: Globalized Nelder-Mead method for engineering optimization. Comput. Struct. **82**(23), 2251–2260 (2004)
31. Nguyen, L.V., Kapinski, J., Jin, X., Deshmukh, J.V., Butts, K., Johnson, T.T.: Abnormal data classification using time-frequency temporal logic. In: Proceedings of the 20th International Conference on Hybrid Systems: Computation and Control, HSCC 2017, pp. 237–242. ACM, New York (2017)
32. Platzer, A.: Logical Foundations of Cyber-Physical Systems. Springer, Cham (2018). <https://doi.org/10.1007/978-3-319-63588-0>
33. Seshia, S.A., Rakhlin, A.: Quantitative analysis of systems using game-theoretic learning. ACM Trans. Embed. Comput. Syst. **11**(S2), 55:1–55:27 (2012)
34. Silvetti, S., Policriti, A., Bortolussi, L.: An active learning approach to the falsification of black box cyber-physical systems. In: Polikarpova, N., Schneider, S. (eds.) IFM 2017. LNCS, vol. 10510, pp. 3–17. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66845-1_1
35. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A.: Experimentation in Software Engineering. Springer, Heidelberg (2012). <https://doi.org/10.1007/978-3-642-29044-2>
36. Zhang, Z., Ernst, G., Sedwards, S., Arcaini, P., Hasuo, I.: Two-layered falsification of hybrid systems guided by monte carlo tree search. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **37**(11), 2894–2905 (2018)
37. Zhang, Z., Hasuo, I., Arcaini, P.: Multi-Armed Bandits for Boolean Connectives in Hybrid System Falsification (Extended Version). CoRR, [arXiv:1905.07549](https://arxiv.org/abs/1905.07549) (2019)
38. Zutshi, A., Deshmukh, J.V., Sankaranarayanan, S., Kapinski, J.: Multiple shooting, CEGAR-based falsification for hybrid systems. In: 2014 International Conference on Embedded Software, EMSOFT 2014, New Delhi, India, 12–17 October 2014, pp. 5:1–5:10. ACM (2014)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





StreamLAB: Stream-based Monitoring of Cyber-Physical Systems

Peter Faymonville, Bernd Finkbeiner,
Malte Schledjewski, Maximilian Schwenger^(✉),
Marvin Stenger, Leander Tentrup,
and Hazem Torfah



Reactive Systems Group, Saarland University,
Saarbrücken, Germany
`{faymonville,finkbeiner,schledjewski,schwenger,
stenger,tentrup,torfah}@react.uni-saarland.de`

Abstract. With ever increasing autonomy of cyber-physical systems, monitoring becomes an integral part for ensuring the safety of the system at runtime. StreamLAB is a monitoring framework with high degree of expressibility and strong correctness guarantees. Specifications are written in RTLola, a stream-based specification language with formal semantics. StreamLAB provides an extensive analysis of the specification, including the computation of memory consumption and run-time guarantees. We demonstrate the applicability of StreamLAB on typical monitoring tasks for cyber-physical systems, such as sensor validation and system health checks.

1 Introduction

In stream-based monitoring, we translate input streams containing data collected at runtime, such as sensor readings, into output streams containing aggregate statistics, such as an average value, a counter, or the integral of a signal. Trigger specifications define thresholds and other logical conditions on the values on these output streams, and raise an alarm or execute some other predefined action if the condition becomes true. The advantage of this setup is great expressiveness and easy-to-reuse, compositional specifications. Existing stream-based languages like Lola [9, 12] are based on the synchronous programming paradigm, where all streams are synchronized via a global clock. In each step, the new values of all output streams are computed in terms of the values of the other streams at a previous time step or. This paradigm provides a simple and natural evaluation model that fits well with typical implementations on

This work was partially supported by the German Research Foundation (DFG) as part of the Collaborative Research Center “Foundations of Perspicuous Software Systems” (TRR 248, 389792660), and by the European Research Council (ERC) Grant OSARES (No. 683300).

© The Author(s) 2019

I. Dillig and S. Tasiran (Eds.): CAV 2019, LNCS 11561, pp. 421–431, 2019.

https://doi.org/10.1007/978-3-030-25540-4_24

synchronous hardware. In real-time applications, however, the assumption that all data arrives synchronously is often simply not true. Consider, for example, an autonomous drone with several sensors, such as a GPS module, an inertia measurement unit, and a laser distance meter. While a synchronous arrival of all measured value would be desirable, some sensors' measurement frequency is higher than others. Moreover, the sensors do not necessarily operate on a common clock, so their readings drift apart over time.

In this paper we present the monitoring framework StreamLAB. We lift the synchronicity assumption to allow for monitoring asynchronous systems. Basis for the framework is RTLola, an extension of the steam-based runtime verification language Lola. RTLola introduces two new key concepts into Lola:

1. Variable-rate input streams: we consider input streams that extend at a-priori unknown rates. The only assumption is that each new event has a real-valued timestamp and that the events arrive in-order.
2. Sliding windows: A sliding window aggregates data over a real-time window given in units of time. For example, we might integrate the readings of an airspeed indicator.

As with any semantic extension, the challenge in the design of RTLola is to maintain the efficiency of the monitoring. Obviously, not all RTLola specifications can be monitored with constant memory since the rates of the input streams are unknown, an arbitrary number of events may occur in the span of a fixed real-time unit. Thus, for aggregations such as the mean requiring to store the whole sequence of value, no amount of constant memory will always suffice. We can, nevertheless, again identify an efficiently monitorable fragment that covers many specifications of practical interest. For the space-efficient aggregation over real-time sliding windows, we partition the real-time axis into equally-sized intervals. The size of the intervals is dictated by the rate of the output streams. For certain common types of aggregations, such as the sum or the number of entries, the values within each interval can be pre-aggregated and then only stored in this summarized form. In a static analysis of the specification, we identify parts of the specification with unbounded memory consumption, and compute bounds for all other parts of the specification. In this way, we can determine early, whether a particular specification can be executed on a system with limited memory.

Related Work. There is a rich body of work on monitoring real-time properties. Many monitoring approaches are based on real-time variants of temporal logics [3, 11, 16–18, 24]. Maler and Nickovic present a monitoring algorithm for properties written in signal temporal logic (STL) by reducing STL formulas via a boolean abstraction to formulas in the real-time logic MITL [21]. Building on these ideas, Donze et al. present an algorithm for the monitoring of STL properties over continuous signals [10]. The algorithm computes the robustness degree in which a piecewise-continuous signal satisfies or violates an STL formula. Towards more practical approaches, Basin et al. extend metric logics with parameterization [8]. A monitoring algorithm for the extension is implemented in the tool MonPoly [5]. MonPoly was introduced as a tool for monitoring usage-control policies. Another extension to metric dynamic logic was implemented in

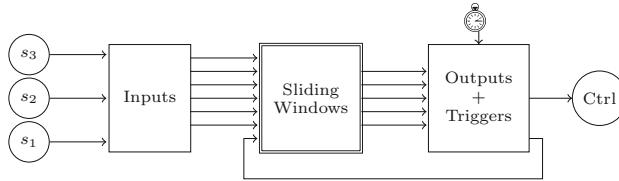


Fig. 1. Illustration of the decoupled input and output using aggregations.

the tool Aerial [7]. However, most monitors generated from temporal logics are limited to Boolean verdicts.

StreamLAB uses the stream-based language RTLola as its core specification language. RTLola builds upon Lola [9, 12], which is a stream-based language originally developed for monitoring synchronous hardware circuits, by adding the concepts discussed above. Stream-based monitoring languages are significantly more expressive than temporal logics. Other prominent stream-based monitoring approaches are the *Copilot* framework [23] and the tool BeepBeep 3 [15]. Copilot is a dataflow language based on several declarative stream processing languages [9, 14]. From a specification in Copilot, constant space and constant time C programs implementing embedded monitors are generated. The BeepBeep 3 tool uses an SQL-like language that is defined over streams of events. In addition to stream-processing, it contains operators such as slicing, where inputs can be separated into several different traces, and windowing where aggregations over a sliding window can be computed. Unlike RTLola, BeepBeep and Copilot assume a synchronous computation model, where all events arrive at a fixed rate. Two asynchronous real-time monitoring approaches are TeSSLa [19] and Striver [13]. TeSSLa allows for monitoring piece-wise constant signals where streams can emit events at different speeds with arbitrary latencies. Neither language provides the language feature of sliding windows and the definition of fixed-rate output streams. The efficient evaluation of aggregations on sliding windows [20] has previously been studied in the context of temporal logic [4]. Basin et al. present an algorithm for combining the elements of subsequences of a sliding window with an associative operator, which reuses the results of the subsequences in the evaluation of the next window [6].

2 Real-Time Lola

RTLola extends the stream-based specification languages Lola [12] with real-time features. In the stream-based processing paradigm, sensor readings are viewed as input streams to a stream processing engine that computes outputs in form of streams on top of the values of the input streams. For example, the RTLola specification

```

input altitude : Float32
output tooLow := altitude < 200.0

```

checks whether a drone flies with an altitude less than 200 feet. For each reading of the velocity sensor, a new value for the output stream *tooLow* is computed. Streams marked with the “**trigger**”-keyword alert the user when the value of the trigger is true. In the following example, the user is warned when the drone flies below the allowed altitude.

```
trigger tooLow "flying below minimum altitude"
```

Output streams in RTLola are computed from values of the input streams, other output streams and their own past values. If we want to count the number of times the drone dives below 200 feet we can specify the stream

```
output count := (if tooLow then 1 else 0)
  + count.offset(by:-1).defaults(to:0)
```

Here, the stream *count* computes its new values by increasing its latest value by 1 in case the drone currently flies below the permitted altitude. The expression *count*.**offset**(by:-1) represents the last value of the stream. We call such expressions “lookup expressions”. The default operator *e*.**defaults**(*to*:0) returns the value 0 in case the value of *e* is not defined. This can happen when a stream is evaluated the first time and looks up its last value.

In RTLola, we do not impose any assumption on the arrival frequency of input streams. Each stream can produce new values individually and at arbitrary points in time. This can lead to problems when a burst of new input values occur in a short amount of time. Subsequently, the monitor needs to evaluate all output streams, exerting a lot of pressure on the system. To prevent that, RTLola distinguishes between two kinds of outputs. *Event-based* outputs are computed whenever new input values arrive and should thus only contain inexpensive operations. All streams discussed above were event-based. In contrast to that, there are *periodic* outputs such as the following:

```
output freqDev @5Hz := altitude.aggregate(over : 200ms,
                                             using: count) < 5
```

Here, *freqDev* will be evaluated every 200 ms as indicated by the “@ 5 Hz” label, independently of arriving input values. The stream *freqDev* does not access the event-based input *altitude* directly, but uses a *sliding window* expression to count the number of times a new value for *altitude* occurred within the last 200 ms. The value of *freqDev* represents the number of measurements the monitor received from the altimeter. Comparing this value against the expected number of readings allows for detecting deviations and thus a potentially damaged sensor.

Sliding windows allow for decoupling event-based and periodic streams, as illustrated in Fig. 1. Since the specifier has no control over the frequency of event-based streams, these streams should be quickly evaluable. More expensive operations, such as sliding windows, may only be used in periodic streams to increase the monitor’s robustness.

2.1 Examples

In the following, we will present several interesting properties showcasing RTLola's expressivity. The specifications are simplified for illustration and thus not immediately applicable to the real-world.

Sensor Validation. When a sensor starts to deteriorate, it can misbehave and drop single measurements. To verify that a GPS sensor produces values at its specified frequency, in this example 10 Hz, we count the number of sensor values in a continuous window and compare it against the expected amount of events in this time frame.

```
input lat: Float32, lon : Float32
output gps_freq@10Hz:=
  lat.aggregate(over: =1s, using: count).defaults(to:9)
trigger gps_freq < 9 "GPS sensor frequency < 9 Hz"
```

Assuming that we have another sensor measuring the true air speed, we can check whether the measured data matches the GPS data using RTLola's computation primitives. For this, we first compute the difference in longitude and latitude between the current and last measurement. The Euclidean distance provides the length of the movement vector, which can be derived discretely by dividing by the amount of time that has passed between two GPS measurements.

```
input velo : Float32
output δlon := lon - lon.offset(by:-1).defaults(to:lon)
output δlat := lat - lat.offset(by:-1).defaults(to:lat)
output gps_dist := sqrt(δlon * δlon + δlat * δlat)
output gps_velo := gps_dist
  / (time - time.offset(by:-1).defaults(to:0.0))
trigger abs(gps_velo - velo) > 0.1 "Deviating velocity"
```

When the pathfinding algorithm of the mission planner takes longer than expected, the system remains in a state without target location and thus hovers in place. Such a hover period can be detected by computing the covered distance in the last seconds. For this, we integrate the assumed velocity. We also exclude a strong headwind as a culprit for the low change in position.

```
input wnd_dir: Float32, wnd_spd : Float32
output dir := arctan(lat/lon)
output headwind := abs(wnd_dir - dir) < 0.2
  ^ wnd_spd > 10.0
output hovering @ 1Hz := velo.aggregate(over: 5s, using: ∫)
  .defaults(to:0.5) < 0.5 ^ ¬headwind.hold().defaults(to:⊥)
trigger hovering "Long hover phase"
```

3 Performance Guarantees via Static Analysis

3.1 Type System

RTLola is a strongly-typed specification language. Every expression has two orthogonal types: a value type and a stream type. The *value* type is `Bool`, `String`, `Int`, or `Float`. It indicates the usual semantics of a value or expression and the amount of memory required to store the value. The *stream* type indicates when a value is evaluated. For periodic streams, the stream type defines the frequency in which it is computed. Event-based streams do not have a pre-determined period. The stream type for an event-based stream identifies a set of input streams, indicating that the event-based stream is extended whenever there is, synchronously, an event on all input streams. Event-based streams may also depend on input streams not listed in the type; in such cases, the type system requires an explicit use of the 0-order `sample&hold` operator.

The type system provides runtime guarantees for the monitor: Independently of the arrival of input data, it is guaranteed that all required data is available whenever a stream is extended. Either the data was just received as input event, was computed as output stream value, or the specifier provided a default value. The type system can, thus, eliminate classes of specification problems like unintentionally accessing a slower stream from a faster stream. Whenever possible, the tool provides automatic type inference.

3.2 Sliding Windows

We use two techniques to ensure that we only need a bounded amount of memory to compute sliding windows. Meertens [22] classifies an aggregations $\gamma: A^* \rightarrow B$ as list homomorphism if it can be split into a mapping function $m: A \rightarrow T$, an associative reduction function $r: T \times T \rightarrow T$, a finalization function $f: T \rightarrow B$, and a neutral element $\varepsilon \in T$ with $\forall t \in T: r(t, \varepsilon) = r(\varepsilon, t) = t$. For these functions, rather than aggregating the whole list at once, one can apply m to each element, reduce the intermediate results with an arbitrary precedence, and finalize the result to get the same value. The second technique by Li et al. [20] divides a time interval into panes of equal size. For each pane, we aggregate all inputs and store only the fix amount of intermediate values. The type system ensures that sliding windows only occur in periodic streams so by choosing the pane size as the inverse of the frequency, paning does not change the result. In StreamLAB there are several pre-defined aggregation functions such as count, integration, summation, product, mini-, and maximization available.

3.3 Memory Analysis

StreamLAB computes the worst-case memory consumption of the specification. For this, an annotated dependency graph (ADG) is constructed where each stream s constitutes a node v_s and whenever s accesses s' , there is an edge from v_s to $v_{s'}$. Edges are annotated according to the type of access: if s accesses

s' discretely with offset n or with a sliding window aggregation of duration d and aggregation function γ , then the edge $e = (v_s, v_{s'})$ is labeled with $\lambda(e) = n$ or $\lambda(e) = (d, \gamma)$, respectively. Nodes of periodic streams are now annotated with their periodicity, if stream s has period 200 ms then the node is labeled with $\pi(v_s) = 5$ Hz. Memory bounds for discrete-time offsets can be computed as for Lola [9]. We extend this algorithm with new computational rules to determine the memory bounds for real-time expressions. For each edge $e = (v, v')$ in the ADG we can determine how many events of v' must be stored for the computation of v using the rules in Fig. 2. Here, only γ is a list homomorphism. The strict upper bound on required memory is now the sum of the memory requirement of each individual stream. This, however, is only the amount of memory needed for storing values and does not take book-keeping data structures and the internal representation of the specification into account. Assuming reasonably small expressions (depth ≤ 64), this additional memory can be bounded with 1 kB per stream plus a flat 10 kB for working memory.

$\pi(v)$	$\pi(v')$	$\lambda(e) = (d, \gamma)$	$\lambda(e) = (d, \gamma^*)$
var	var	$unbounded$	zd
xHz	var	$unbounded$	$\min(zd, xd)$
var	yHz	yd	$\min(zd, yd)$
xHz	yHz	$\min(xd, yd)$	$\min(xd, yd)$

Fig. 2. Computation of memory bound over the dependency graph.

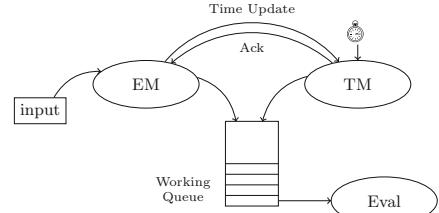


Fig. 3. Illustration of the data flow. The EM manages input events, TM schedules periodic tasks, and Eval manages the evaluation of streams.

4 Processing Engine

The processing engine consists of three components: The *EventManager* (*EM*) reads events from an input such Standard In or a CSV file and translates string values into the internal representation. The values are mapped to the corresponding input streams in the specification. Using a multiple-sender-single-receiver channel, the EM pushes the event on a working queue. The *TimeManager* (*TM*) schedules the evaluation of periodic streams. The TM computes the hyper-period of all streams and groups them by equal deadlines. Whenever a deadline is due, the corresponding streams are pushed into the working queue using the same channel as the EM. This ensures that event-based and periodic evaluation cycles occur in the correct order even under high pressure. Lastly, the *Evaluator* (*Eval*) manages the evaluation of streams and storage of computed values. The Eval repeatedly pops items off the working queue and evaluates the respective streams.

When monitoring a system online, the TM uses the internal system clock for scheduling tasks. When monitoring offline, however, this is no longer possible because the point in time when a stream is due to be evaluated depends on the input event. Thus, before the EM pushes an event on the working queue, it transmits the latest timestamp to the TM. The TM then decides whether some periodic streams need to be evaluated. If so, it effectively goes back in time by pushing the respective task on the working queue before acknowledging the TM. Only upon receiving the acknowledgement, the TM sends the event to the working queue. Figure 3 illustrates the information flow between the components.

5 Experiments

StreamLAB¹ is implemented in Rust. A major benefit of a Rust implementation is the connection to LLVM, which allows a compilation to a large variety of platforms. Moreover, the requirements to the runtime environment are as low as for C programs. This allows StreamLAB to be widely applicable.

The specifications presented in Sect. 2.1 have been tested on traces generated with the state-of-the-art flight simulator ARDUPILOT². Each trace is the result of a drone flying one or more round-trips over Saarland University and provides sensor information for longitude and latitude, true air velocity, wind direction and speed, as well as the number of available GPS satellites. The longest trace consists of slightly less than 433,000 events. StreamLAB successfully detected a variety of errors such as delayed sensor readings, GPS module failures, and phases without significant movement. For an online runtime verification, the monitor reads an event of the simulator’s output, processes the input data and pauses until the next event is available. Whenever necessary, periodic streams are evaluated. Online monitoring of a simulation did not allow us to exhaust the capabilities of StreamLAB because the generation of events took significantly longer than processing them. The offline monitoring function of StreamLAB allows the user to specify a delay in which consecutive events are read from a file. By gradually decreasing the delay between events until the pressure was too high, we could determine a maximum input frequency of 647.2 kHz. When disabling the delay and running the monitor at maximum speed, StreamLAB processes a trace of length 432,961 in 0.67 s, so each event takes 1545 ns to process while three threads utilized 146% of CPU. In terms of memory, the maximum resident set size amounted to 16 MB. This includes bookkeeping data structures, the specification, evaluator code, and parts of the C standard library. While the evaluation does not require any heap allocation after the setup phase, the average stack size amounts to less than 1 kB. The experiment was conducted on 3.3 GHz Intel Core i7 processor with 16 GB 2133 MHz LPDDR3 RAM.

¹ www.stream-lab.org.

² ardupilot.org.

6 Outlook

The stream-based monitoring framework StreamLAB demonstrates the applicability of stream monitoring for cyber-physical systems. Previous versions of Lola have successfully been applied to networks and unmanned aircraft systems in cooperation with German Aerospace Center DLR [1, 2, 12]. StreamLAB provides a modular, easy-to-understand specification language and design-time feedback for specifiers. This helps to improve the development process for cyber-physical systems. Coupled with the promising experimental results, this lays the foundation for further applications of the framework on real-world systems.

References

1. Adolf, F.-M., Faymonville, P., Finkbeiner, B., Schirmer, S., Torens, C.: Stream runtime monitoring on UAS. In: Lahiri, S., Reger, G. (eds.) RV 2017. LNCS, vol. 10548, pp. 33–49. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67531-2_3
2. Adolf, F., Faymonville, P., Finkbeiner, B., Schirmer, S., Torens, C.: Stream runtime monitoring on UAS. CoRR [arXiv:abs/1804.04487](https://arxiv.org/abs/1804.04487) (2018). <http://arxiv.org/abs/1804.04487>
3. Alur, R., Henzinger, T.A.: Real-time logics: complexity and expressiveness. In: [1990] Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science, pp. 390–401, June 1990. <https://doi.org/10.1109/LICS.1990.113764>
4. Basin, D., Bhatt, B.N., Traytel, D.: Almost event-rate independent monitoring of metric temporal logic. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10206, pp. 94–112. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54580-5_6
5. Basin, D., Harvan, M., Klaedtke, F., Zălinescu, E.: MONPOLY: monitoring usage-control policies. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 360–364. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29860-8_27
6. Basin, D., Klaedtke, F., Zălinescu, E.: Greedily computing associative aggregations on sliding windows. Inf. Process. Lett. **115**(2), 186–192 (2015). <https://doi.org/10.1016/j.ipl.2014.09.009>
7. Basin, D., Traytel, D., Krstić, S.: Aerial: almost event-rate independent algorithms for monitoring metric regular properties (2017). https://www21.in.tum.de/~traytel/papers/rvcubes17-aerial_tool/index.html
8. Basin, D.A., Klaedtke, F., Müller, S., Zalinescu, E.: Monitoring metric first-order temporal properties. J. ACM **62**(2), 15:1–15:45 (2015). <https://doi.org/10.1145/2699444>
9. D’Angelo, B., et al.: Lola: Runtime monitoring of synchronous systems. In: TIME 2005, pp. 166–174. IEEE Computer Society Press, June 2005
10. Donzé, A., Ferrère, T., Maler, O.: Efficient robust monitoring for STL. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 264–279. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_19
11. Donzé, A., Maler, O.: Robust satisfaction of temporal logic over real-valued signals. In: Chatterjee, K., Henzinger, T.A. (eds.) FORMATS 2010. LNCS, vol. 6246, pp. 92–106. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15297-9_9. <http://dl.acm.org/citation.cfm?id=1885174.1885183>

12. Faymonville, P., Finkbeiner, B., Schirmer, S., Torfah, H.: A stream-based specification language for network monitoring. In: Falcone, Y., Sánchez, C. (eds.) RV 2016. LNCS, vol. 10012, pp. 152–168. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46982-9_10
13. Gorostiaga, F., Sánchez, C.: Striver: stream runtime verification for real-time event-streams. In: Colombo, C., Leucker, M. (eds.) RV 2018. LNCS, vol. 11237, pp. 282–298. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03769-7_16
14. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous dataflow programming language lustre. In: Proceedings of the IEEE, pp. 1305–1320 (1991)
15. Hallé, S.: When RV meets CEP. In: Falcone, Y., Sánchez, C. (eds.) RV 2016. LNCS, vol. 10012, pp. 68–91. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46982-9_6
16. Harel, E., Lichtenstein, O., Pnueli, A.: Explicit clock temporal logic. In: LICS 1990, pp. 402–413. IEEE Computer Society (1990). <https://doi.org/10.1109/LICS.1990.113765>
17. Jahanian, F., Mok, A.K.L.: Safety analysis of timing properties in real-time systems. IEEE Trans. Softw. Eng. **SE-12**(9), 890–904 (1986). <https://doi.org/10.1109/TSE.1986.6313045>
18. Koymans, R.: Specifying real-time properties with metric temporal logic. Real-Time Syst. **2**(4), 255–299 (1990). <https://doi.org/10.1007/BF01995674>
19. Leucker, M., Sánchez, C., Scheffel, T., Schmitz, M., Schramm, A.: Tessla: runtime verification of non-synchronized real-time streams. In: Haddad, H.M., Wainwright, R.L., Chbeir, R. (eds.) PSAC 2018, pp. 1925–1933. ACM (2018). <https://doi.org/10.1145/3167132.3167338>
20. Li, J., Maier, D., Tufte, K., Papadimos, V., Tucker, P.A.: No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. SIGMOD Rec. **34**(1), 39–44 (2005). <https://doi.org/10.1145/1058150.1058158>
21. Maler, O., Nickovic, D.: Monitoring temporal properties of continuous signals. In: Lakhnech, Y., Yovine, S. (eds.) FORMATS/FTRTFT -2004. LNCS, vol. 3253, pp. 152–166. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30206-3_12
22. Meertens, L.: Algorithmics: towards programming as a mathematical activity (1986)
23. Pike, L., Goodloe, A., Morisset, R., Niller, S.: Copilot: a hard real-time runtime monitor. In: Barringer, H., et al. (eds.) RV 2010. LNCS, vol. 6418, pp. 345–359. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16612-9_26
24. Raskin, J.-F., Schobbens, P.-Y.: Real-time logics: fictitious clock as an abstraction of dense time. In: Brinksma, E. (ed.) TACAS 1997. LNCS, vol. 1217, pp. 165–182. Springer, Heidelberg (1997). <https://doi.org/10.1007/BFb0035387>

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





VERIFAI: A Toolkit for the Formal Design and Analysis of Artificial Intelligence-Based Systems

Tommaso Dreossi^(✉), Daniel J. Fremont^(✉),
Shromona Ghosh^(✉), Edward Kim, Hadi Ravanbakhsh,
Marcell Vazquez-Chanlatte, and Sanjit A. Seshia^(✉)



University of California, Berkeley, USA
{tommasodreossi, dfremont, shromona.ghosh}@berkeley.edu,
ssesha@eecs.berkeley.edu

Abstract. We present VERIFAI, a software toolkit for the formal design and analysis of systems that include artificial intelligence (AI) and machine learning (ML) components. VERIFAI particularly addresses challenges with applying formal methods to ML components such as perception systems based on deep neural networks, as well as systems containing them, and to model and analyze system behavior in the presence of environment uncertainty. We describe the initial version of VERIFAI, which centers on simulation-based verification and synthesis, guided by formal models and specifications. We give examples of several use cases, including temporal-logic falsification, model-based systematic fuzz testing, parameter synthesis, counterexample analysis, and data set augmentation.

Keywords: Formal methods · Falsification · Simulation ·
Cyber-physical systems · Machine learning · Artificial intelligence ·
Autonomous vehicles

1 Introduction

The increasing use of artificial intelligence (AI) and machine learning (ML) in systems, including safety-critical systems, has brought with it a pressing need for formal methods and tools for their design and verification. However, AI/ML-based systems, such as autonomous vehicles, have certain characteristics that make the application of formal methods very challenging. We mention three key challenges here; see Seshia et al. [23] for an in-depth discussion. First, several uses of AI/ML are for *perception*, the use of computational systems to mimic human perceptual tasks such as object recognition and classification, conversing in natural language, etc. For such perception components,

This work was supported in part by NSF grants 1545126 (VeHICaL), 1646208, 1739816, and 1837132, the DARPA BRASS program under agreement number FA8750-16-C0043, the DARPA Assured Autonomy program, the iCyPhy center, and Berkeley Deep Drive. NVIDIA Corporation donated the Titan Xp GPU used for this research.

T. Dreossi, D. J. Fremont, S. Ghosh—These authors contributed equally to the paper.

© The Author(s) 2019
I. Dillig and S. Tasiran (Eds.): CAV 2019, LNCS 11561, pp. 432–442, 2019.
https://doi.org/10.1007/978-3-030-25540-4_25

writing a formal specification is extremely difficult, if not impossible. Additionally, the signals processed by such components can be very high-dimensional, such as streams of images or LiDAR data. Second, *machine learning* being a dominant paradigm in AI, formal tools must be compatible with the data-driven design flow for ML and also be able to handle the complex, high-dimensional structures in ML components such as deep neural networks. Third, the *environments* in which AI/ML-based systems operate can be very complex, with considerable uncertainty even about how many (which) agents are in the environment (both human and robotic), let alone about their intentions and behaviors. As an example, consider the difficulty in modeling urban traffic environments in which an autonomous car must operate. Indeed, AI/ML is often introduced into these systems precisely to deal with such complexity and uncertainty! From a formal methods perspective, this makes it very hard to create realistic environment models with respect to which one can perform verification or synthesis.

In this paper, we introduce the VERIFAI toolkit, our initial attempt to address the three core challenges—perception, learning, and environments—that are outlined above. VERIFAI takes the following approach:

- *Perception*: A perception component maps a concrete feature space (e.g. pixels) to an output such as a classification, prediction, or state estimate. To deal with the lack of specification for perception components, VERIFAI analyzes them in the context of a closed-loop system using a system-level specification. Moreover, to scale to complex high-dimensional feature spaces, VERIFAI operates on an *abstract feature space* (or *semantic feature space*) [10] that describes semantic aspects of the environment being perceived, not the raw features such as pixels.
- *Learning*: VERIFAI aims to not only analyze the behavior of ML components but also use formal methods for their (re-)design. To this end, it provides features to (i) design the data set for training and testing [9], (ii) analyze counterexamples to gain insight into mistakes by the ML model, as well as (iii) synthesize parameters, including hyper-parameters for training algorithms and ML model parameters.
- *Environment Modeling*: Since it can be difficult, if not impossible, to exhaustively model the environments of AI-based systems, VERIFAI aims to provide ways to capture a designer’s assumptions about the environment, including distribution assumptions made by ML components, and to describe the abstract feature space in an intuitive, declarative manner. To this end, VERIFAI provides users with SCENIC [12, 13], a probabilistic programming language for modeling environments. SCENIC, combined with a renderer or simulator for generating sensor data, can produce semantically-consistent input for perception components.

VERIFAI is currently focused on AI-based cyber-physical systems (CPS), although its basic ideas can also be applied to other AI-based systems. As a pragmatic choice, we focus on simulation-based verification, where the simulator is treated as a black-box, so as to be broadly applicable to the range of simulators used in industry.¹ The input to

¹ Our work is complementary to the work on industrial-grade simulators for AI/ML-based CPS. In particular, VERIFAI enhances such simulators by providing formal methods for modeling (via the SCENIC language), analysis (via temporal logic falsification), and parameter synthesis (via property-directed hyper/model-parameter synthesis).

VERIFAI is a “closed-loop” CPS model, comprising a composition of the AI-based CPS system under verification with an environment model, and a property on the closed-loop model. The AI-based CPS typically comprises a perception component (not necessarily based on ML), a planner/controller, and the plant (i.e., the system under control). Given these, VERIFAI offers the following use cases: (1) temporal-logic falsification; (2) model-based fuzz testing; (3) counterexample-guided data augmentation; (4) counterexample (error table) analysis; (5) hyper-parameter synthesis, and (6) model parameter synthesis. The novelty of VERIFAI is that it is the first tool to offer this suite of use cases in an integrated fashion, unified by a common representation of an abstract feature space, with an accompanying modeling language and search algorithms over this feature space, all provided in a modular implementation. The algorithms and formalisms in VERIFAI are presented in papers published by the authors in other venues (e.g., [7–10, 12, 15, 22]). The problem of temporal-logic falsification or simulation-based verification of CPS models is well studied and several tools exist (e.g. [3, 11]); our work was the first to extend these techniques to CPS models with ML components [7, 8]. Work on verification of ML components, especially neural networks (e.g., [14, 26]), is complementary to the system-level analysis performed by VERIFAI. Fuzz testing based on formal models is common in software engineering (e.g. [16]) but our work is unique in the CPS context. Similarly, property-directed parameter synthesis has also been studied in the formal methods/CPS community, but our work is the first to apply these ideas to the synthesis of hyper-parameters for ML training and ML model parameters. Finally, to our knowledge, our work on augmenting training/test data sets [9], implemented in VERIFAI, is the first use of formal techniques for this purpose. In Sect. 2, we describe how the tool is structured so as to provide the above features. Sect. 3 illustrates the use cases via examples from the domain of autonomous driving.

2 VERIFAI Structure and Operation

VERIFAI is currently focused on simulation-based analysis and design of AI components for perception or control, potentially those using ML, in the context of a closed-loop cyber-physical system. Figure 1 depicts the structure and operation of the toolkit.

Inputs and Outputs: Using VERIFAI requires setting up a simulator for the domain of interest. As we explain in Sect. 3, we have experimented with multiple robotics simulators and provide an easy interface to connect a new simulator. The user then constructs the inputs to VERIFAI, including (i) a simulatable model of the system, including code for one or more controllers and perception components, and a dynamical model of the system being controlled; (ii) a probabilistic model of the environment, specifying constraints on the workspace, the locations of agents and objects, and the dynamical behavior of agents, and (iii) a property over the composition of the system and its environment. VERIFAI is implemented in Python for interoperability with ML/AI libraries and simulators across platforms. The code for the controller and perception component can be arbitrary executable code, invoked by the simulator. The environment model typically comprises a definition in the simulator of the different types of agents, plus a description of their initial conditions and other parameters using the SCENIC probabilistic programming language [12]. Finally, the property to be checked can be expressed

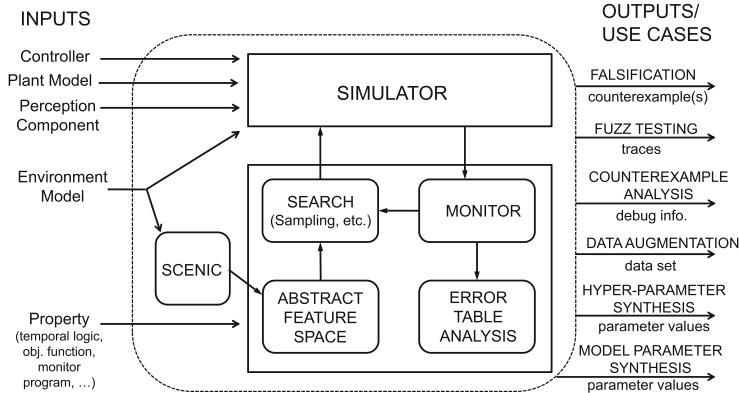


Fig. 1. Structure and operation of VERIFAI.

using Metric Temporal Logic (MTL) [2, 24], objective functions, or arbitrary code monitoring the property. The output of VERIFAI depends on the feature being invoked. For falsification, VERIFAI returns one or more *counterexamples*, simulation traces violating the property [7]. For fuzz testing, VERIFAI produces traces sampled from the distribution of behaviors induced by the probabilistic environment model [12]. Error table analysis involves collecting counterexamples generated by the falsifier into a table, on which we perform analysis to identify features that are correlated with property failures. Data augmentation uses falsification and error table analysis to generate additional data for training and testing an ML component [9]. Finally, the property-driven synthesis of model parameters or hyper-parameters generates as output a parameter evaluation that satisfies the specified property.

Tool Structure: VERIFAI is composed of four main modules, as described below:

- *Abstract Feature Space and SCENIC Modeling Language:* The abstract feature space is a compact representation of the possible configurations of the simulation. Abstract features can represent parameters of the environment, controllers, or of ML components. For example, when analyzing a visual perception system for an autonomous car, an abstract feature space could consist of the initial poses and types of all vehicles on the road. Note that this abstract space, compared to the concrete feature space of pixels used as input to the controller, is better suited to the analysis of the overall closed-loop system (e.g. finding conditions under which the car might crash).

VERIFAI provides two ways to construct abstract feature spaces. They can be constructed hierarchically, combining basic domains such as hyperboxes and finite sets into structures and arrays. For example, we could define a space for a car as a structure combining a 2D box for position with a 1D box for heading, and then create an array of these to get a space for several cars. Alternatively, VERIFAI allows a feature space to be defined using a program in the SCENIC language [12]. SCENIC provides convenient syntax for describing geometric configurations and agent parameters, and, as a probabilistic programming language, allows placing a distribution over the feature space which can be conditioned by declarative constraints.

- *Searching the Feature Space:* Once the abstract feature space is defined, the next step is to search that space to find simulations that violate the property or produce other interesting behaviors. Currently, VERIFAI uses a suite of sampling methods (both active and passive) for this purpose, but in the future we expect to also integrate directed or exhaustive search methods including those from the adversarial machine learning literature (e.g., see [10]). Passive samplers, which do not use any feedback from the simulation, include uniform random sampling, simulated annealing, and Halton sequences [18] (quasi-random deterministic sequences with low-discrepancy guarantees we found effective for falsification [7]). Distributions defined using SCENIC are also passive in this sense. Active samplers, whose selection of samples is informed by feedback from previous simulations, include cross-entropy sampling and Bayesian optimization. The former selects samples and updates the prior distribution by minimizing cross-entropy; the latter updates the prior from the posterior over a user-provided objective function, e.g. the satisfaction level of a specification or the loss of an analyzed model.
- *Property Monitor:* Trajectories generated by the simulator are evaluated by the monitor, which produces a score for a given property or objective function. VERIFAI supports monitoring MTL properties using the `py-metric-temporal-logic` [24] package, including both the Boolean and quantitative semantics of MTL. As mentioned above, the user can also specify a custom monitor as a Python function. The result of the monitor can be used to output falsifying traces and also as feedback to the search procedure to direct the sampling (search) towards falsifying scenarios.
- *Error Table Analysis:* Counterexamples are stored in a data structure called the error table, whose rows are counterexamples and columns are abstract features. The error table can be used offline to debug (explain) the generated counterexamples or online to drive the sampler towards particular areas of the abstract feature space. VERIFAI provides different techniques for error table analysis depending on the end use (e.g., counter-example analysis or data set augmentation), including principal component analysis (PCA) for ordered feature domains and subsets of the most recurrent values for unordered domains (see [9] for further details).

The communication between VERIFAI and the simulator is implemented in a client-server fashion using IPv4 sockets, where VERIFAI sends configurations to the simulator which then returns trajectories (traces). This architecture allows easy interfacing to a simulator and even with multiple simulators at the same time.

3 Features and Case Studies

This section illustrates the main features of VERIFAI through case studies demonstrating its various use cases and simulator interfaces. Specifically, we demonstrate model falsification and fuzz testing of an autonomous vehicle (AV) controller, data augmentation and error table analysis for a convolutional neural network, and model and hyperparameter tuning for a reinforcement learning-based controller.

3.1 Falsification and Fuzz Testing

VERIFAI offers a convenient way to debug systems through systematic testing. Given a model and a specification, the tool can use active sampling to automatically search for inputs driving the model towards a violation of the specification. VERIFAI can also perform model-based fuzz testing, exploring random variations of a scenario guided by formal constraints. To demonstrate falsification and fuzz testing, we consider two scenarios involving AVs simulated with the robotics simulator Webots [25]. For the experiments reported here, we used Webots 2018 which is commercial software.

In the first example, we falsify the controller of an AV which is responsible for safely maneuvering around a disabled car and traffic cones which are blocking the road. We implemented a hybrid controller which relies on perception modules for state estimation. Initially, the car follows its lane using standard computer vision (non-ML) techniques for line detection [20]. At the same time, a neural network (based on squeezeDet [27]) estimates the distance to the cones. When the distance drops below 15 m, the car performs a lane change, afterward switching back to lane-following.

The correctness of the AV is characterized by an MTL formula requiring the vehicle to maintain a minimum distance from the traffic cones and avoid overshoot while changing lanes. The task of the falsifier is to find small perturbations of the initial scene (generated by SCENIC) which cause the vehicle to violate this specification. We allowed perturbations of the initial positions and orientations of all objects, the color of the disabled car, and the cruising speed and reaction time of the ego car.

Our experiments showed that active samplers driven by the robustness of the MTL specification can efficiently discover scenes that confuse the controller and yield faulty behavior. Figure 2 shows an example, where the neural network detected the orange car instead of the traffic cones, causing the lane change to be initiated too early. As a result, the controller performed only an incomplete lane change, leading to a crash.

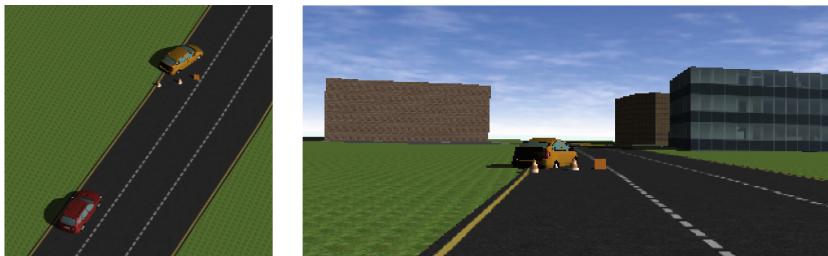


Fig. 2. A falsifying scene automatically discovered by VERIFAI. The neural network misclassifies the traffic cones because of the orange vehicle in the background, leading to a crash. Left: bird's-eye view. Right: dash-cam view, as processed by the neural network.

In our second experiment, we used VERIFAI to simulate variations on an actual accident involving an AV [5]. The AV, proceeding straight through an intersection, was hit by a human turning left. Neither car was able to see the other because of two lanes of stopped traffic. Figure 3 shows a (simplified) SCENIC program we wrote to reproduce

```

# Car going straight
ego = Car on egoLane.median

# Car turning left
Car on leftTurnLane.median

# A car blocking the Ego's view
spot = OrientedPoint on blockLane.median
laneNoise = (-0.5, 0.5)
Car at spot offset by laneNoise @ 0

# Another car 5-8 m behind that
Car at spot2 offset by laneNoise @ (-5, -8)

```

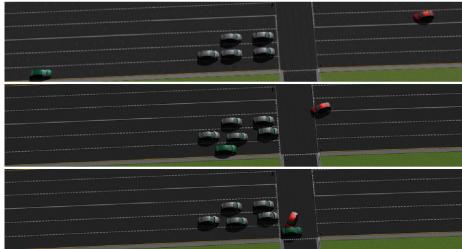


Fig. 3. Left: Partial SCENIC program for the crash scenario. `Car` is an object class defined in the Webots world model (not shown), `on` is a SCENIC *specifier* positioning the object uniformly at random in the given region (e.g. the median line of a lane), $(-0.5, 0.5)$ indicates a uniform distribution over that interval, and $X @ Y$ creates a vector with the given coordinates (see [12] for a complete description of SCENIC syntax). Right: (1) initial scene sampled from the program; (2) the red car begins its turn, unable to see the green car; (3) the resulting collision. (Color figure online)

the accident, allowing variation in the initial positions of the cars. We then ran simulations from random initial conditions sampled from the program, with the turning car using a controller trying to follow the ideal left-turn trajectory computed from OpenStreetMap data using the Intelligent Intersections Toolbox [17]. The car going straight used a controller which either maintained a constant velocity or began emergency braking in response to a message from a simulated “smart intersection” warning about the turning car. By sampling variations on the initial conditions, we could determine how much advance notice is necessary for such a system to robustly avoid an accident.

3.2 Data Augmentation and Error Table Analysis

Data augmentation is the process of supplementing training sets with the goal of improving the performance of ML models. Typically, datasets are augmented with transformed versions of preexisting training examples. In [9], we showed that augmentation with counterexamples is also an effective method for model improvement.

VERIFAI implements a counterexample-guided augmentation scheme, where a falsifier (see Sect. 3.1) generates misclassified data points that are then used to augment the original training set. The user can choose among different sampling methods, with passive samplers suited to generating diverse sets of data points while active samplers can efficiently generate similar counterexamples. In addition to the counterexamples themselves, VERIFAI also returns an error table aggregating information on the misclassifications that can be used to drive the retraining process. Figure 4 shows the rendering of a misclassified sample generated by our falsifier.



Fig. 4. This image generated by our renderer was misclassified by the NN. The network reported detecting only one car when there were two.

For our experiments, we implemented a renderer that generates images of road scenarios and tested the quality of our augmentation scheme on the squeezeDet convolutional neural network [27], trained for classification. We adopted three techniques to select augmentation images: (1) randomly sampling from the error table, (2) selecting the top k -closest (similar) samples from the error table, and (3) using PCA analysis to generate new samples. For details on the renderer and the results of counterexample-driven augmentation, see [9]. We show that incorporating the generated counterexamples during re-training improves the accuracy of the network.

3.3 Model Robustness and Hyperparameter Tuning

In this final section, we demonstrate how VERIFAI can be used to tune test parameters and hyperparameters of AI systems. For the following case studies, we use OpenAI Gym [4], a framework for experimenting with reinforcement learning algorithms.

First, we consider the problem of testing the robustness of a learned controller for a cart-pole, i.e., a cart that balances an inverted pendulum. We trained a neural network to control the cart-pole using Proximal Policy Optimization algorithms [21] with 100k training episodes. We then used VERIFAI to test the robustness of the learned controller, varying the initial lateral position and rotation of the cart as well as the mass and length of the pole. Even for apparently robust controllers, VERIFAI was able to discover configurations for which the cart-pole failed to self-balance. Figure 5 shows 1000 iterations of the falsifier, where sampling was guided by the reward function used by OpenAI to train the controller. This function provides a negative reward if the cart moves more than 2.4 m or if at any time the angle maintained by the pole is greater than 12°. For testing, we slightly modified these thresholds.

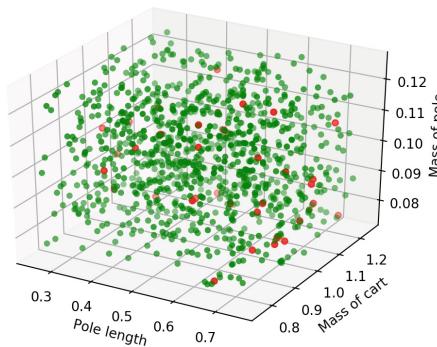


Fig. 5. The green dots represent model parameters for which the cart-pole controller behaved correctly, while the red dots indicate specification violations. Out of 1000 randomly-sampled model parameters, the controller failed to satisfy the specification 38 times. (Color figure online)

Finally, we used VERIFAI to study the effects of hyperparameters when training a neural network controller for a mountain car. In this case, the controller must learn to

exploit momentum in order to climb a steep hill. Here, rather than searching for counterexamples, we look for a set of hyperparameters under which the network *correctly* learns to control the car. Specifically, we explored the effects of using different training algorithms (from a discrete set of choices) and the size of the training set. We used the VERIFAI falsifier to search the hyperparameter space, guided again by the reward function provided by OpenAI Gym (here the distance from the goal position), but negated so that falsification implied finding a controller which successfully climbs the hill. In this way VERIFAI built a table of safe hyperparameters. PCA analysis then revealed which hyperparameters the training process is most sensitive or robust to.

4 Conclusion

We presented VERIFAI, a toolkit for the formal design and analysis of AI/ML-based systems. Our implementation, plus the examples described in Sect. 3, are available in the tool distribution [1], including detailed instructions and expected output.

In future work, we plan to explore additional applications of VERIFAI, and to expand its functionality with new algorithms. Towards the former, we have already interfaced VERIFAI to the CARLA driving simulator [6], for more sophisticated experiments with autonomous cars, as well as to the X-Plane flight simulator [19], for testing an ML-based aircraft navigation system. More broadly, although our focus has been on CPS, we note that VERIFAI’s architecture is applicable to other types of systems. Finally, for extending VERIFAI itself, we plan to move beyond directed simulation by incorporating symbolic methods, such as those used in finding adversarial examples.

References

1. VerifAI: a toolkit for the design and analysis of artificial intelligence-based systems. <https://github.com/BerkeleyLearnVerify/VerifAI>
2. Alur, R., Henzinger, T.A.: Logics and models of real time: a survey. In: de Bakker, J.W., Huizing, C., de Roever, W.P., Rozenberg, G. (eds.) REX 1991. LNCS, vol. 600, pp. 74–106. Springer, Heidelberg (1992). <https://doi.org/10.1007/BFb0031988>
3. Annpureddy, Y., Liu, C., Fainekos, G.E., Sankaranarayanan, S.: S-taliro: a tool for temporal logic falsification for hybrid systems. In: Tools and Algorithms for the Construction and Analysis of Systems, TACAS (2011)
4. Brockman, G., et al.: OpenAI Gym. [arXiv:1606.01540](https://arxiv.org/abs/1606.01540) (2016)
5. Butler, M.: Uber’s tempe accident raises questions of self-driving safety. East Valley Tribune (2017). http://www.eastvalleytribune.com/local/tempe/uber-s-tempe-accident-raises-questions-of-self-driving-safety/article_30b99e74-189d-11e7-bc1d-07f943301a72.html
6. Dosovitskiy, A., Ros, G., Codevilla, F., Lopez, A., Koltun, V.: CARLA: an open urban driving simulator. In: Conference on Robot Learning, CoRL, pp. 1–16 (2017)
7. Dreossi, T., Donzé, A., Seshia, S.A.: Compositional falsification of cyber-physical systems with machine learning components. In: Barrett, C., Davies, M., Kahsai, T. (eds.) NFM 2017. LNCS, vol. 10227, pp. 357–372. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-57288-8_26
8. Dreossi, T., Donze, A., Seshia, S.A.: Compositional falsification of cyber-physical systems with machine learning components. J. Autom. Reasoning (JAR) (2019)

9. Dreossi, T., Ghosh, S., Yue, X., Keutzer, K., Sangiovanni-Vincentelli, A., Seshia, S.A.: Counterexample-guided data augmentation. In: 27th International Joint Conference on Artificial Intelligence (IJCAI) (2018)
10. Dreossi, T., Jha, S., Seshia, S.A.: Semantic adversarial deep learning. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 3–26. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_1
11. Duggirala, P.S., Mitra, S., Viswanathan, M., Potok, M.: C2E2: a verification tool for stateflow models. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 68–82. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_5
12. Fremont, D.J., Dreossi, T., Ghosh, S., Yue, X., Sangiovanni-Vincentelli, A.L., Seshia, S.A.: Scenic: a language for scenario specification and scene generation. In: 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) (2019, to appear)
13. Fremont, D.J., Yue, X., Dreossi, T., Ghosh, S., Sangiovanni-Vincentelli, A.L., Seshia, S.A.: Scenic: language-based scene generation. CoRR (2018). [arXiv:1809.09310](https://arxiv.org/abs/1809.09310)
14. Gehr, T., Mirman, M., Drachsler-Cohen, D., Tsankov, P., Chaudhuri, S., Vechev, M.: AI2: safety and robustness certification of neural networks with abstract interpretation. In: 2018 IEEE Symposium on Security and Privacy (SP) (2018)
15. Ghosh, S., Berkenkamp, F., Ranade, G., Qadeer, S., Kapoor, A.: Verifying controllers against adversarial examples with Bayesian optimization. In: 2018 IEEE International Conference on Robotics and Automation (ICRA) (2018)
16. Godefroid, P., Kiezun, A., Levin, M.Y.: Grammar-based whitebox fuzzing. In: ACM SIGPLAN Notices. ACM (2008)
17. Grembek, O., Kurzhanskiy, A.A., Medury, A., Varaiya, P., Yu, M.: Making intersections safer with I2V communication (2019). [arXiv:1803.00471](https://arxiv.org/abs/1803.00471), to appear in Transportation Research, Part C
18. Halton, J.H.: On the efficiency of certain quasi-random sequences of points in evaluating multi-dimensional integrals. Numer. Math. **2**, 84–90 (1960)
19. Laminar Research: X-Plane 11 (2019). <https://www.x-plane.com/>
20. Palazzi, A.: Finding lane lines on the road (2018). https://github.com/ndrplz/self-driving-car/tree/master/project_1_lane_finding_basic
21. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O.: Proximal policy optimization algorithms. CoRR (2017). [arXiv:1707.06347](https://arxiv.org/abs/1707.06347)
22. Seshia, S.A., et al.: Formal specification for deep neural networks. In: Lahiri, S.K., Wang, C. (eds.) ATVA 2018. LNCS, vol. 11138, pp. 20–34. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-01090-4_2
23. Seshia, S.A., Sadigh, D., Sastry, S.S.: Towards Verified Artificial Intelligence. CoRR (2016). [arXiv:1606.08514](https://arxiv.org/abs/1606.08514)
24. Vazquez-Chanlatte, M.: mvcisback/py-metric-temporal-logic: v0.1.1 (2019). <https://doi.org/10.5281/zenodo.2548862>
25. Webots: Commercial mobile robot simulation software. <http://www.cyberbotics.com>
26. Wicker, M., Huang, X., Kwiatkowska, M.: Feature-guided black-box safety testing of deep neural networks. In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10805, pp. 408–426. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89960-2_22
27. Wu, B., Iandola, F., Jin, P.H., Keutzer, K.: SqueezeDet: Unified, small, low power fully convolutional neural networks for real-time object detection for autonomous driving. In: CVPR 2017 (2016). <https://doi.org/10.1109/CVPRW.2017.60>

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





The Marabou Framework for Verification and Analysis of Deep Neural Networks

Guy Katz^{1(✉)}, Derek A. Huang², Dugigur Ibeling²,
Kyle Julian², Christopher Lazarus², Rachel Lim²,
Parth Shah², Shantanu Thakoor², Haoze Wu²,
Aleksandar Zeljić², David L. Dill²,
Mykel J. Kochenderfer², and Clark Barrett²

¹ The Hebrew University of Jerusalem,
Jerusalem, Israel

guykatz@cs.huji.ac.il

² Stanford University, Stanford, USA

{huangda,dugigur,kjulian3,clazarus,parth95,thakoor,
haozewu,zeljic,dill,mykel,clarkbarrett}@stanford.edu,
rachelim@cs.stanford.edu



Abstract. Deep neural networks are revolutionizing the way complex systems are designed. Consequently, there is a pressing need for tools and techniques for network analysis and certification. To help in addressing that need, we present *Marabou*, a framework for verifying deep neural networks. Marabou is an SMT-based tool that can answer queries about a network’s properties by transforming these queries into constraint satisfaction problems. It can accommodate networks with different activation functions and topologies, and it performs high-level reasoning on the network that can curtail the search space and improve performance. It also supports parallel execution to further enhance scalability. Marabou accepts multiple input formats, including protocol buffer files generated by the popular TensorFlow framework for neural networks. We describe the system architecture and main components, evaluate the technique and discuss ongoing work.

1 Introduction

Recent years have brought about a major change in the way complex systems are being developed. Instead of spending long hours hand-crafting complex software, many engineers now opt to use *deep neural networks (DNNs)* [6, 19]. DNNs are machine learning models, created by training algorithms that generalize from a finite set of examples to previously unseen inputs. Their performance can often surpass that of manually created software as demonstrated in fields such as image classification [16], speech recognition [8], and game playing [21].

Despite their overall success, the opacity of DNNs is a cause for concern, and there is an urgent need for certification procedures that can provide rigorous guarantees about network behavior. The formal methods community has

taken initial steps in this direction, by developing algorithms and tools for neural network verification [5, 9, 10, 12, 18, 20, 23, 24]. A DNN verification query consists of two parts: (i) a neural network, and (ii) a property to be checked; and its result is either a formal guarantee that the network satisfies the property, or a concrete input for which the property is violated (a counter-example). A verification query can encode the fact, e.g., that a network is robust to small adversarial perturbations in its input [22].

A neural network is comprised of *neurons*, organized in layers. The network is evaluated by assigning values to the neurons in the input layer, and then using these values to iteratively compute the assignments of neurons in each succeeding layer. Finally, the values of neurons in the last layer are computed, and this is the network’s output. A neuron’s assignment is determined by computing a weighted sum of the assignments of neurons from the preceding layer, and then applying to the result a non-linear activation function, such as the Rectified Linear Unit (ReLU) function, $\text{ReLU}(x) = \max(0, x)$. Thus, a network can be regarded as a set of *linear constraints* (the weighted sums), and a set of *non-linear constraints* (the activation functions). In addition to a neural network, a verification query includes a property to be checked, which is given in the form of linear or non-linear constraints on the network’s inputs and outputs. The verification problem thus reduces to finding an assignment of neuron values that satisfies all the constraints simultaneously, or determining that no such assignment exists.

This paper presents a new tool for DNN verification and analysis, called *Marabou*. The Marabou project builds upon our previous work on the Reluplex project [2, 7, 12, 13, 15, 17], which focused on applying SMT-based techniques to the verification of DNNs. Marabou follows the Reluplex spirit in that it applies an SMT-based, *lazy search* technique: it iteratively searches for an assignment that satisfies all given constraints, but treats the non-linear constraints lazily in the hope that many of them will prove irrelevant to the property under consideration, and will not need to be addressed at all. In addition to search, Marabou performs deduction aimed at learning new facts about the non-linear constraints in order to simplify them.

The Marabou framework is a significant improvement over its predecessor, Reluplex. Specifically, it includes the following enhancements and modifications:

- Native support for fully connected and convolutional DNNs with arbitrary piecewise-linear activation functions. This extends the Reluplex algorithm, which was originally designed to support only ReLU activation functions.
- Built-in support for a *divide-and-conquer* solving mode, in which the solver is run with an initial (small) timeout. If the timeout is reached, the solver partitions its input query into simpler sub-queries, increases the timeout value, and repeats the process on each sub-query. This mode naturally lends itself to parallel execution by running sub-queries on separate nodes; however, it can yield significant speed-ups even when used with a single node.
- A complete simplex-based linear programming core that replaces the external solver (GLPK) that was previously used in Reluplex. The new simplex

core was tailored for a smooth integration with the Marabou framework and eliminates much of the overhead in Reluplex due to the use of GLPK.

- Multiple interfaces for feeding queries into the solver. A query’s neural network can be provided in a textual format or as a protocol buffer (*protobuf*) file containing a TensorFlow model; and the property can be either compiled into the solver, provided in Python, or stored in a textual format. We expect these interfaces will simplify usage of the tool for many users.
- Support for network-level reasoning and deduction. The earlier Reluplex tool performed deductions at the level of single constraints, ignoring the input network’s topology. In Marabou, we retain this functionality but also include support for reasoning based on the network topology, such as symbolic bound tightening [23]. This allows for efficient curtailment of the search space.

Marabou is available online [14] under the permissive modified BSD license.

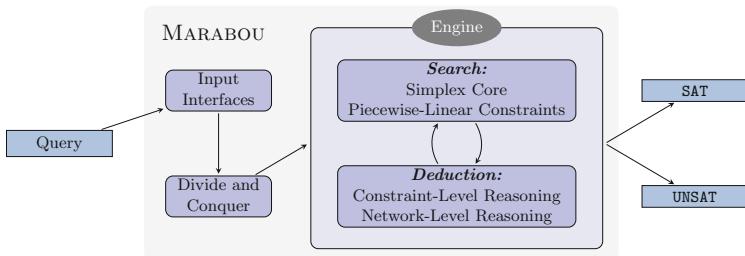


Fig. 1. The main components of Marabou.

2 Design of Marabou

Marabou regards each neuron in the network as a variable and searches for a variable assignment that simultaneously satisfies the query’s linear constraints and non-linear constraints. At any given point, Marabou maintains the current variable assignment, lower and upper bounds for every variable, and the set of current constraints. In each iteration, it then changes the variable assignment in order to (1) correct a violated linear constraint, or (2) correct a violated non-linear constraint.

The Marabou verification procedure is sound and complete, i.e. the aforementioned loop eventually terminates. This can be shown via a straightforward extension of the soundness and completeness proof for Reluplex [12]. However, in order to guarantee termination, Marabou only supports activation functions that are piecewise-linear. The tool already has built-in support for the ReLU function and the Max function $\max(x_1, \dots, x_n)$, and it is modular in the sense that additional piecewise-linear functions can be added easily.

Another important aspect of Marabou’s verification strategy is deduction—specifically, the derivation of tighter lower and upper variable bounds. The motivation is that such bounds may transform piecewise-linear constraints into linear constraints, by restricting them to one of their linear segments. To achieve this, Marabou repeatedly examines linear and non-linear constraints, and also performs network-level reasoning, with the goal of discovering tighter variable bounds.

Next, we describe Marabou’s main components (see also Fig. 1).

2.1 Simplex Core (*Tableau* and *BasisFactorization* Classes)

The simplex core is the part of the system responsible for making the variable assignments satisfy the linear constraints. It does so by implementing a variant of the *simplex algorithm* [3]. In each iteration, it changes the assignment of some variable x , and consequently the assignment of any variable y that is connected to x by a linear equation. Selecting x and determining its new assignment is performed using standard algorithms—specifically, the *revised simplex method* in which the various linear constraints are kept in implicit matrix form, and the steepest-edge and Harris’ ratio test strategies for variable selection.

Creating an efficient simplex solver is complicated. In Reluplex, we delegated the linear constraints to an external solver, GLPK. Our motivation for implementing a new custom solver in Marabou was twofold: first, we observed in Reluplex that the repeated translation of queries into GLPK and extraction of results from GLPK was a limiting factor on performance; and second, a black box simplex solver did not afford the flexibility we needed in the context of DNN verification. For example, in a standard simplex solver, variable assignments are typically pressed against their upper or lower bounds, whereas in the context of a DNN, other assignments might be needed to satisfy the non-linear constraints. Another example is the deduction capability, which is crucial for efficiently verifying a DNN and whose effectiveness might depend on the internal state of the simplex solver.

2.2 Piecewise-Linear Constraints (*PiecewiseLinearConstraint* Class)

Throughout its execution, Marabou maintains a set of piecewise-linear constraints that represent the DNN’s non-linear functions. In iterations devoted to satisfying these constraints, Marabou looks for any constraints that are not satisfied by the current assignment. If such a constraint is found, Marabou changes the assignment in a way that makes that constraint satisfied. Alternatively, in order to guarantee eventual termination, if Marabou detects that a certain constraint is repeatedly not satisfied, it may perform a *case-split* on that constraint: a process in which the piecewise-linear constraint φ is replaced by an equivalent disjunction of linear constraints $c_1 \vee \dots \vee c_n$. Marabou considers these disjuncts one at a time and checks for satisfiability. If the problem is satisfiable when φ is

replaced by some c_i , then the original problem is also satisfiable; otherwise, the original problem is unsatisfiable.

In our implementation, piecewise-linear constraints are represented by objects of classes that inherit from the *PiecewiseLinearConstraint* abstract class. Currently the two supported instances are ReLU and Max, but the design is modular in the sense that new constraint types can easily be added. *PiecewiseLinearConstraint* defines the interface methods that each supported piecewise-linear constraint needs to implement. Some of the key interface methods are:

- *satisfied()*: the constraint object needs to answer whether or not it is satisfied given the current assignment. For example, for a constraint $y = \text{ReLU}(x)$ and assignment $x = y = 3$, *satisfied()* would return *true*; whereas for assignment $x = -5, y = 3$, it would return *false*.
- *getPossibleFixes()*: if the constraint is not satisfied by the current assignment, this method returns possible changes to the assignment that would correct the violation. For example, for $x = -5, y = 3$, the ReLU constraint from before might propose two possible changes to the assignment, $x \leftarrow 3$ or $y \leftarrow 0$, as either would satisfy $y = \text{ReLU}(x)$.
- *getCaseSplits()*: this method asks the piecewise-linear constraint φ to return a list of linear constraints c_1, \dots, c_n , such that φ is equivalent to $c_1 \vee \dots \vee c_n$. For example, when invoked for a constraint $y = \max(x_1, x_2)$, *getCaseSplits()* would return the linear constraints $c_1 : (y = x_1 \wedge x_1 \geq x_2)$ and $c_2 : (y = x_2 \wedge x_2 \geq x_1)$. These constraints satisfy the requirement that the original constraint is equivalent to $c_1 \vee c_2$.
- *getEntailedTightenings()*: as part of Marabou’s deduction of tighter variable bounds, piecewise-linear constraints are repeatedly informed of changes to the lower and upper bounds of variables they affect. Invoking *getEntailedTightenings()* queries the constraint for tighter variable bounds, based on current information. For example, suppose a constraint $y = \text{ReLU}(x)$ is informed of the upper bounds $x \leq 5$ and $y \leq 7$; in this case, *getEntailedTightenings()* would return the tighter bound $y \leq 5$.

2.3 Constraint- and Network-Level Reasoning (*RowBoundTightener*, *ConstraintBoundTightener* and *SymbolicBoundTightener* Classes)

Effective deduction of tighter variable bounds is crucial for Marabou’s performance. Deduction is performed at the constraint level, by repeatedly examining linear and piecewise-linear constraints to see if they imply tighter variable bounds; and also at the DNN-level, by leveraging the network’s topology.

Constraint-level bound tightening is performed by querying the piecewise-linear constraints for tighter bounds using the *getEntailedTightenings()* method. Similarly, linear equations can also be used to deduce tighter bounds. For example, the equation $x = y + z$ and lower bounds $x \geq 0, y \geq 1$ and $z \geq 1$ together imply the tighter bound $x \geq 2$. As part of the simplex-based search, Marabou repeatedly encounters many linear equations and uses them for bound tightening.

Several recent papers have proposed verification schemes that rely on DNN-level reasoning [5, 23]. Marabou supports this kind of reasoning as well, by storing the initial network topology and performing deduction steps that use this information as part of its iterative search. DNN-level reasoning is seamlessly integrated into the search procedure by (1) initializing the DNN-level reasoners with the most up-to-date information discovered during the search, such as variable bounds and the state of piecewise-linear constraints; and (2) feeding any new information that is discovered back into the search procedure. Presently Marabou implements a symbolic bound tightening procedure [23]: based on network topology, upper and lower bounds for each hidden neuron are expressed as a linear combination of the input neurons. Then, if the bounds on the input neurons are sufficiently tight (e.g., as a result of past deductions), these expressions for upper and lower bounds may imply that some of the hidden neurons' piecewise-linear activation functions are now restricted to one of their linear segments. Implementing additional DNN-level reasoning operations is work in progress.

2.4 The Engine (*Engine* and *SmtCore* Classes)

The main class of Marabou, in which the main loop resides, is called the *Engine*. The engine stores and coordinates the various solution components, including the simplex core and the piecewise-linear constraints. The main loop consists, roughly, of the following steps (the first rule that applies is used):

1. If a piecewise-linear constraint had to be fixed more than a certain number of times, perform a case split on that constraint.
2. If the problem has become unsatisfiable, e.g. because for some variable a lower bound has been deduced that is greater than its upper bound, undo a previous case split (or return `UNSAT` if no such case split exists).
3. If there is a violated linear constraint, perform a simplex step.
4. If there is a violated piecewise-linear constraint, attempt to fix it.
5. Return `SAT` (all constraints are satisfied).

The engine also triggers deduction steps, both at the neuron level and at the network level, according to various heuristics.

2.5 The Divide-and-Conquer Mode and Concurrency (*DnC.py*)

Marabou supports a *divide-and-conquer* (*D&C*) solving mode, in which the input region specified in the original query is partitioned into sub-regions. The desired property is checked on these sub-regions independently. The D&C mode naturally lends itself to parallel execution, by having each sub-query checked on a separate node. Moreover, the D&C mode can improve Marabou's overall performance even when running sequentially: the total time of solving the sub-queries is often less than the time of solving the original query, as the smaller input regions allow for more effective deduction steps.

Given a query ϕ , the solver maintains a queue Q of $\langle \text{query}, \text{timeout} \rangle$ pairs. Q is initialized with one element $\langle \phi, T \rangle$, where T , the initial timeout, is a configurable parameter. To solve ϕ , the solver loops through the following steps:

1. Pop a pair $\langle \phi', t' \rangle$ from Q and attempt to solve ϕ' with a timeout of t' .
2. If the problem is **UNSAT** and Q is empty, return **UNSAT**.
3. If the problem is **UNSAT** and Q is not empty, return to step 1.
4. If the problem is **SAT**, return **SAT**.
5. If a timeout occurred, split ϕ' into k sub-queries ϕ'_1, \dots, ϕ'_k by partitioning its input region. For each sub-query ϕ'_i , push $\langle \phi'_i, m \cdot t' \rangle$ into Q .

The timeout factor m and the splitting factor k are configurable parameters. Splitting the query's input region is performed heuristically.

2.6 Input Interfaces (*AcasParser* class, *maraboupy* Folder)

Marabou supports verification queries provided through the following interfaces:

- Native Marabou format: a user prepares a query using the Marabou C++ interface, compiles the query into the tool, and runs it. This format is useful for integrating Marabou into a larger framework.
- Marabou executable: a user runs a Marabou executable, and passes to it command-line parameters indicating the network and property files to be checked. Currently, network files are encoded using the *NNet* format [11], and the properties are given in a simple textual format.
- Python/TensorFlow interface: the query is passed to Marabou through Python constructs. The python interface can also handle DNNs stored as TensorFlow protobuf files.

3 Evaluation

For our evaluation we used the ACAS Xu [12], CollisionDetection [4] and TwinStream [1] families of benchmarks. Tool-wise, we considered the Reluplex tool which is the most closely related to Marabou, and also ReluVal [23] and Planet [4]. The version of Marabou used for the evaluation is available online [14].

The top left plot in Fig. 3 compares the execution times of Marabou and Reluplex on 180 ACAS Xu benchmarks with a 1 hour timeout. We used Marabou in D&C mode with 4 cores and with $T = 5$, $k = 4$, and $m = 1.5$. The remaining three plots depict an execution time comparison between Marabou D&C (configuration as above), ReluVal and Planet, using 4 cores and a 1 hour timeout. Marabou and Reluval are evaluated over 180 ACAS Xu benchmarks (top right plot), and Marabou and Planet are evaluated on those 180 benchmarks (bottom left plot) and also on 500 CollisionDetection and 81 TwinStream benchmarks (bottom right plot). Due to technical difficulties, ReluVal was not run on the CollisionDetection and TwinStream benchmarks. The results show that in a 4 cores setting Marabou generally outperforms Planet, but generally does not outperform ReluVal (though it does better on some benchmarks). These results highlight the need for additional DNN-level reasoning in Marabou, which is a key ingredient in ReluVal's verification procedure.

Figure 2 shows the average runtime of Marabou and ReluVal on the ACAS Xu properties, as a function of the number of available cores. We see that as the number of cores increases, Marabou (solid) is able to close the gap, and sometimes outperform, ReluVal (dotted). With 64 cores, Marabou outperforms ReluVal on average, and both solvers were able to solve all ACAS Xu benchmarks within 2 hours (except for a few segfaults by ReluVal).

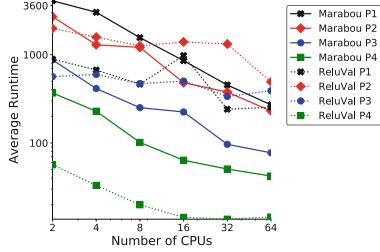


Fig. 2. A scalability comparison of Marabou and ReluVal on ACAS Xu.

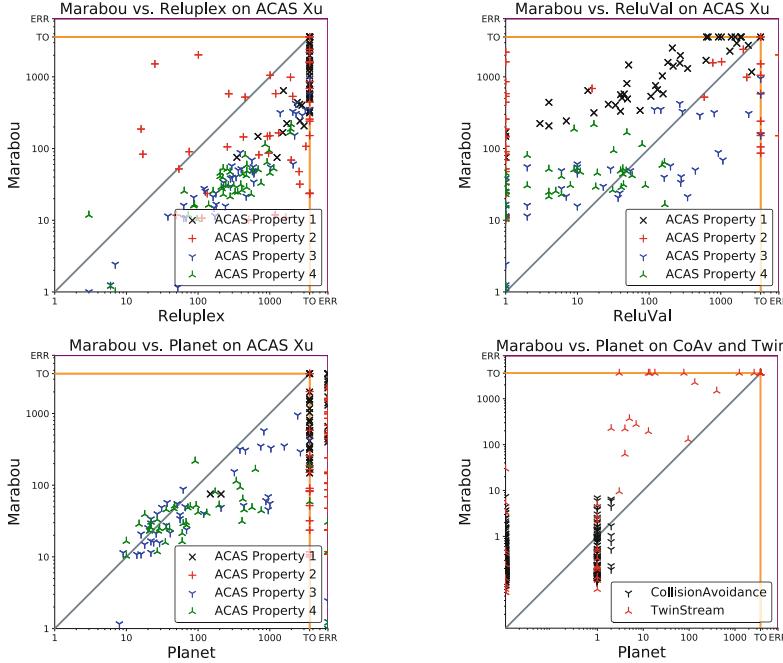


Fig. 3. A comparison of Marabou with Reluplex, ReluVal and Planet.

4 Conclusion

DNN analysis is an emerging field, and Marabou is a step towards a more mature, stable verification platform. Moving forward, we plan to improve Marabou in several dimensions. Part of our motivation in implementing a custom simplex solver was to obtain the needed flexibility for fusing together the solving process for linear and non-linear constraints. Currently, this flexibility has not been leveraged much, as these pieces are solved relatively separately. We expect that by tackling

both kinds of constraints simultaneously, we will be able to improve performance significantly. Other enhancements we wish to add include: additional network-level reasoning techniques based on abstract interpretation; better heuristics for both the linear and non-linear constraint solving engines; and additional engineering improvements, specifically within the simplex engine.

Acknowledgements. We thank Elazar Cohen, Justin Gottschlich, and Lindsey Kuper for their contributions to this project. The project was partially supported by grants from the Binational Science Foundation (2017662), the Defense Advanced Research Projects Agency (FA8750-18-C-0099), the Federal Aviation Administration, Ford Motor Company, Intel Corporation, the Israel Science Foundation (683/18), the National Science Foundation (1814369, DGE-1656518), Siemens Corporation, and the Stanford CURIS program.

References

1. Bunel, R., Turkaslan, I., Torr, P., Kohli, P., Kumar, M.: Piecewise linear neural network verification: a comparative study. Technical report (2017). [arXiv:1711.00455v1](https://arxiv.org/abs/1711.00455v1)
2. Carlini, N., Katz, G., Barrett, C., Dill, D.: Provably minimally-distorted adversarial examples. Technical report (2017). [arXiv:1709.10207](https://arxiv.org/abs/1709.10207)
3. Chvátal, V.: Linear Programming. W. H. Freeman and Company, New York (1983)
4. Ehlers, R.: Formal verification of piece-wise linear feed-forward neural networks. In: D’Souza, D., Narayan Kumar, K. (eds.) ATVA 2017. LNCS, vol. 10482, pp. 269–286. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68167-2_19
5. Gehr, T., Mirman, M., Drachsler-Cohen, D., Tsankov, E., Chaudhuri, S., Vechev, M.: AI2: safety and robustness certification of neural networks with abstract interpretation. In: Proceedings of 39th IEEE Symposium on Security and Privacy (S&P) (2018)
6. Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning. MIT Press, Cambridge (2016)
7. Gopinath, D., Katz, G., Păsăreanu, C., Barrett, C.: DeepSafe: a data-driven approach for checking adversarial robustness in neural networks. In: Proceedings of 16th International Symposium on on Automated Technology for Verification and Analysis (ATVA), pp. 3–19 (2018)
8. Hinton, G., et al.: Deep neural networks for acoustic modeling in speech recognition: the shared views of four research groups. IEEE Signal Process. Mag. **29**(6), 82–97 (2012)
9. Huang, X., Kwiatkowska, M., Wang, S., Wu, M.: Safety verification of deep neural networks. In: Proceedings of 29th International Conference on Computer Aided Verification (CAV), pp. 3–29 (2017)
10. Hull, J., Ward, D., Zakrzewski, R.: Verification and validation of neural networks for safety-critical applications. In: Proceedings of 21st American Control Conference (ACC) (2002)
11. Julian, K.: NNet Format (2018). <https://github.com/sisl/NNet>
12. Katz, G., Barrett, C., Dill, D.L., Julian, K., Kochenderfer, M.J.: Reluplex: an efficient SMT solver for verifying deep neural networks. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 97–117. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_5

13. Katz, G., Barrett, C., Dill, D., Julian, K., Kochenderfer, M.: Towards proving the adversarial robustness of deep neural networks. In: Proceedings of 1st Workshop on Formal Verification of Autonomous Vehicles (FVAV), pp. 19–26 (2017)
14. Katz, G., et al.: Marabou (2019). https://github.com/guykatzz/Marabou/tree/cav_artifact
15. Kazak, Y., Barrett, C., Katz, G., Schapira, M.: Verifying deep-RL-driven systems. In: Proceedings of 1st ACM SIGCOMM Workshop on Network Meets AI & ML (NetAI) (2019)
16. Krizhevsky, A., Sutskever, I., Hinton, G.: ImageNet classification with deep convolutional neural networks. In: Advances in Neural Information Processing Systems, pp. 1097–1105 (2012)
17. Kuper, L., Katz, G., Gottschlich, J., Julian, K., Barrett, C., Kochenderfer, M.: Toward scalable verification for safety-critical deep networks. Technical report (2018). [arXiv:1801.05950](https://arxiv.org/abs/1801.05950)
18. Pulina, L., Tacchella, A.: An abstraction-refinement approach to verification of artificial neural networks. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 243–257. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_24
19. Riesenhuber, M., Tomaso, P.: Hierarchical models of object recognition in cortex. Nat. Neurosci. **2**(11), 1019–1025 (1999)
20. Ruan, W., Huang, X., Kwiatkowska, M.: Reachability analysis of deep neural networks with provable guarantees. In: Proceedings of 27th International Joint Conference on Artificial Intelligence (IJCAI) (2018)
21. Silver, D., et al.: Mastering the game of go with deep neural networks and tree search. Nature **529**(7587), 484–489 (2016)
22. Szegedy, C., et al.: Intriguing properties of neural networks. Technical report (2013). [arXiv:1312.6199](https://arxiv.org/abs/1312.6199)
23. Wang, S., Pei, K., Whitehouse, J., Yang, J., Jana, S.: Formal security analysis of neural networks using symbolic intervals. Technical report (2018). [arXiv:1804.10829](https://arxiv.org/abs/1804.10829)
24. Xiang, W., Tran, H., Johnson, T.: Output reachable set estimation and verification for multi-layer neural networks. IEEE Trans. Neural Netw. Learn. Syst. (TNNLS) **99**, 1–7 (2018)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.



Probabilistic Systems, Runtime Techniques



Probabilistic Bisimulation for Parameterized Systems

(with Applications to Verifying Anonymous Protocols)

Chih-Duo Hong^{1(✉)}, Anthony W. Lin^{2(✉)}, Rupak Majumdar^{3(✉)},
and Philipp Rümmer^{4(✉)}

¹ Oxford University, Oxford, UK

chihduo.hong@gmail.com

² TU Kaiserslautern, Kaiserslautern, Germany

lin@cs.uni-kl.de

³ Max Planck Institute for Software Systems, Kaiserslautern, Germany

rupak@mpi-sws.org

⁴ Uppsala University, Uppsala, Sweden

philipp.ruemmer@it.uu.se

Abstract. Probabilistic bisimulation is a fundamental notion of process equivalence for probabilistic systems. It has important applications, including the formalisation of the anonymity property of several communication protocols. While there is a large body of work on verifying probabilistic bisimulation for finite systems, the problem is in general undecidable for parameterized systems, i.e., for infinite families of finite systems with an arbitrary number n of processes. In this paper we provide a general framework for reasoning about probabilistic bisimulation for parameterized systems. Our approach is in the spirit of software verification, wherein we encode proof rules for probabilistic bisimulation and use a decidable first-order theory to specify systems and candidate bisimulation relations, which can then be checked automatically against the proof rules.

We work in the framework of regular model checking, and specify an infinite-state system as a regular relation described by a first-order formula over a universal automatic structure, i.e., a logical theory over the string domain. For probabilistic systems, we show how probability values (as well as the required operations) can be encoded naturally in the logic. Our main result is that one can specify the verification condition of whether a given regular binary relation is a probabilistic bisimulation as a regular relation. Since the first-order theory of the universal automatic structure is decidable, we obtain an effective method for verifying probabilistic bisimulation for infinite-state systems, given a regular relation as a candidate proof. As a case study, we show that our framework is sufficiently expressive for proving the anonymity property of the parameterized dining cryptographers protocol and the parameterized grades protocol. Both of these protocols hitherto could not be verified by existing automatic methods.

This research was sponsored in part by the ERC Starting Grant 759969 (AV-SMP), ERC Synergy project 610150 (ImPACT), the DFG project 389792660-TRR 248 (Perspicuous Computing), the Swedish Research Council (VR) under grant 2018-04727, and by the Swedish Foundation for Strategic Research (SSF) under the project WebSec (Ref. RIT17-0011).

© The Author(s) 2019

I. Dillig and S. Tasiran (Eds.): CAV 2019, LNCS 11561, pp. 455–474, 2019.

https://doi.org/10.1007/978-3-030-25540-4_27

Moreover, with the help of standard automata learning algorithms, we show that the candidate relations can be synthesized fully automatically, making the verification fully automated.

1 Introduction

Equivalence checking using bisimulation relations plays a fundamental role in formal verification. Bisimulation is the basis for substitutability of systems: if two systems are bisimilar, their behaviors are the same and they satisfy the same formulas in expressive temporal logics. The notion of bisimulation is defined both for deterministic [39] and for probabilistic transition systems [34]. In both contexts, checking bisimulation has many applications, such as proving correctness of anonymous communication protocols [15], reasoning about knowledge [22], program optimization [32], and optimizations for computational problems (e.g. language equivalence and minimization) of finite automata [12].

The problem of checking bisimilarity of two given systems has been widely studied. It is decidable in polynomial-time for both probabilistic and non-probabilistic *finite-state* systems [6, 17, 20, 52]. These algorithms form the basis of practical tools for checking bisimulation. For infinite-state systems, such as parameterized versions of communication protocols (i.e. infinite families of finite-state systems with an arbitrary number n of processes), the problem is undecidable in general. Most research hitherto has focused on identifying decidable subcases (e.g. strong bisimulations for pushdown systems for probabilistic and non-probabilistic cases [25, 47, 48]), rather than on providing tool support for practical problems.

In this paper, we propose a first-order verification approach—inspired by software verification techniques—for reasoning about bisimilarity for infinite-state systems. In our approach, we provide first-order logic *proof rules* to determine if a given binary relation is a bisimulation. To this end, we must find an *encoding* of systems and relations and a *decidable first-order theory* that can formalize the system, the property, and the proof rules. We propose to use the decidable first-order theory of the *universal automatic structure* [8, 10]. Informally, the domain of the theory is a set of words over a finite alphabet Σ , and it captures the first-order theory of the infinite $|\Sigma|$ -ary tree with a relation that relates strings of the same level. The theory can express precisely the class of all *regular relations* [8] (a.k.a. automatic relations [10]), which are relations $\varphi(x_1, \dots, x_k)$ over strings Σ^* that can be recognized by synchronous multi-tape automata. It is also sufficiently powerful to capture many classes of non-probabilistic infinite-state systems and regular model checking [3, 13, 49–51].

We demonstrate the effectiveness of the approach by encoding and automatically verifying some challenging examples from the literature of parameterized systems in our logic: the anonymity property of the parameterized dining cryptographers protocol [16] and the grades protocol [29]. These examples were only automatically verified for some fixed parameters using finite-state model checkers or equivalence checkers (e.g. see [28, 29]). Just as invariant verification for software separates out the proof rules (verification conditions in a decidable logic) from the synthesis of invariants, we separate out proof rules for bisimulation from the synthesis of bisimulation relations.

We demonstrate how recent developments in generating and refining candidate proofs as automata (e.g. [18, 26, 27, 37, 38, 40, 41, 53]) can be used to automate the search of proofs, making our verification fully “push button.”

Contributions. Our contributions are as follows. First, we show how probabilistic infinite-state systems can be faithfully encoded in the first-order theory of universal automatic structure. In the past, the theory has been used to reason about qualitative liveness of weakly-finite MDPs (e.g. see [36, 37]), which allows the authors to disregard the actual non-zero probability values. To the best of our knowledge, no encoding of probabilistic transition systems in the theory was available. In order to be able to effectively encode probabilistic systems, our theory should typically be two-sorted: one sort for encoding the configurations, and the other for encoding the probability values. We show how both sorts (and the operations required for the sorts) can be encoded in the universal automatic structure, which requires only the domain of strings. In the sequel, such transition systems will be called *regular transition systems*.

Second, using the *minimal probability assumption* on the transition systems [34] (i.e. there exists an $\varepsilon > 0$ such that any non-zero transition probability is at least ε)—which is often satisfied in practice—we show how the verification condition of whether a given regular binary relation is a probabilistic bisimulation can be encoded in the theory. The decidability of the first-order theory over the universal automatic structure gives us an effective means of checking probabilistic bisimulation for regular transition systems. In fact, the theory can be easily reduced to the weak monadic theory WS1S of one successor (therefore, allowing highly optimized tools like Mona [31] and Gaston [23]) by interpreting finite words as finite sets (e.g. see [19, 46]).

Our framework requires the encoding of the systems and the proofs in the first-order theory of the universal automatic structure. Which interesting examples can it capture? Our third contribution is to provide two examples from the literature of parameterized verification: the anonymity property of the parameterized dining cryptographers protocol [16] and of the parameterized grades protocol [29]. We study two versions of dining cryptographers protocol in this paper: the classical version where the secrets are single bits, and a generalized version where the secrets are bit-vectors of arbitrary length.

Thus far, our framework requires a candidate proof to be supplied by the user. Our final contribution is to demonstrate how standard techniques from the synthesis literature (e.g. automata learning [18, 26, 27, 37, 38, 40, 41, 53]) can be used to fully automate the proof search. Using automata learning, we successfully pinpoint regular proofs for the anonymity property of the three protocols: the two dining cryptographers protocols are verified in 6 and 28 s, respectively, and the grades protocol in 35 s.

Other Related Work. The verification framework we use in this paper can be construed as a regular model checking [3] framework using regular relations. The framework uses first-order logic as the language, which makes it convenient to express many verification conditions (as is well-known from first-order theorem proving [14]). The use of the universal automatic structure allows us to express two different sorts (configurations and probability values) in one sort (i.e. strings). Most work in regular model checking focuses on safety and liveness properties (e.g. [2, 3, 11, 13, 27, 36, 37, 40, 42, 49, 51, 53]).

Some automated techniques can prove the anonymity property of the dining cryptographers protocol and the grades protocol in the finite case, e.g., the PRISM model

checker [28,45] and language equivalence by the tool APEX [29]. To the best of our knowledge, our method is the first automated technique proving the anonymity property of the protocols in the parameterized case.

Our work is in spirit of deductive software verification (e.g., [4, 14, 24, 35, 43, 44]), where one provides inductive invariants manually, and a tool automatically checks correctness of the candidate invariants. In theory, our result yields a fully-automatic procedure by enumerating all candidate regular proofs, and at the same time enumerating all candidate counterexamples (note that we avoid undecidability by restricting attention to proofs encodable as regular relations). In our implementation, we use recent advances in automata-learning based synthesis to efficiently encode the search [18,37].

2 Preliminaries

General Notation. We use \mathbb{N} to denote non-negative integers. Given $a, b \in \mathbb{R}$, we use a standard notation $[a, b] := \{c \in \mathbb{R} : a \leq c \leq b\}$ to denote real intervals. Given a set S , we use S^* to denote the set of all finite sequences of elements from S . The set S^* always includes the empty sequence which we denote by ε . We call a function $f : S \rightarrow [0, 1]$ a *probability distribution over S* if $\sum_{s \in S} f(s) = 1$. We shall use I_s to denote the probability distribution f with $f(s) = 1$, and \mathcal{D}_S to denote the set of probability distributions over S . Given a function $f : X_1 \times \dots \times X_n \rightarrow Y$, the *graph* of f is the relation $\{(x_1, \dots, x_n, f(x_1, \dots, x_n)) : \forall i \in \{1, \dots, n\}. x_i \in X_i\}$. Whenever a relation R is an equivalence relation over set S , we use S/R to denote the set of equivalence classes created by R . Depending on the context, we may use $p R q$ or $R(p, q)$ to denote $(p, q) \in R$.

Words and Automata. We assume basic familiarity with word automata. Fix a finite alphabet Σ . For each finite word $w := w_1 \dots w_n \in \Sigma^*$, we write $w[i, j]$, where $1 \leq i \leq j \leq n$, to denote the segment $w_i \dots w_j$. Given an automaton $\mathcal{A} := (\Sigma, Q, \delta, q_0, F)$, a run of \mathcal{A} on w is a function $\rho : \{0, \dots, n\} \rightarrow Q$ with $\rho(0) = q_0$ that obeys the transition relation δ . We may also denote the run ρ by the word $\rho(0) \dots \rho(n)$ over the alphabet Q . The run ρ is said to be *accepting* if $\rho(n) \in F$, in which case we say that the word w is *accepted* by \mathcal{A} . The language $L(\mathcal{A})$ of \mathcal{A} is the set of words in Σ^* accepted by \mathcal{A} .

Transition Systems. We fix a set ACT of *action symbols*. A *transition system* over ACT is a tuple $\mathfrak{S} := \langle S; \{\rightarrow_a\}_{a \in \text{ACT}} \rangle$, where S is a set of *configurations* and $\rightarrow_a \subseteq S \times S$ is a binary relation over S . We use \rightarrow to denote the relation $\bigcup_{a \in \text{ACT}} \rightarrow_a$. We say that a sequence $s_1 \rightarrow \dots \rightarrow s_{n+1}$ is a *path* in \mathfrak{S} if $s_1, \dots, s_{n+1} \in S$ and $s_i \rightarrow s_{i+1}$ for $i \in \{1, \dots, n\}$. A transition system is called *bounded branching* if the number of configurations reachable from a configuration in one step is bounded. Formally, this means that there exists an *a priori* integer N such that for all $s \in S$, $|\{s' \in S : s \rightarrow s'\}| \leq N$.

Probabilistic Transition Systems. A *probabilistic transition system* (PTS) [34] is a structure $\mathfrak{S} := \langle S; \{\delta_a\}_{a \in \text{ACT}} \rangle$ where S is a set of configurations and $\delta_a : S \rightarrow \mathcal{D}_S \cup \{\bar{0}\}$ maps each configuration to either a probability distribution or a zero function $\bar{0}$. Here $\delta_a(s) = \bar{0}$ simply means that s is a “dead end” for action a . We shall use $\delta_a(s, s')$ to denote $\delta_a(s)(s')$. In this paper, we always assume that $\delta_a(s, s')$ is a rational number

and $|\{s' : \delta_a(s, s') \neq 0\}| < \infty$. The *underlying transition graph* of a PTS is a transition system $\langle S; \{\rightarrow_a\}_{a \in \text{ACT}} \rangle$ such that $s \rightarrow_a s'$ iff $\delta_a(s, s') \neq 0$.

It is standard (e.g. see [34]) to impose the *minimal probability assumption* on the PTS that we shall be dealing with, i.e., there is $\epsilon > 0$ such that any transition with a non-zero probability p satisfies $p > \epsilon$. This assumption is practically sensible since it is satisfied by most PTSs that we deal with in practice (e.g. finite PTS, probabilistic pushdown automata [21], and most examples from probabilistic parameterized systems [36,37] including our examples from Sect. 5). The minimal probability assumption, among others, implies that the PTS is bounded-branching (i.e. that its underlying transition graph is bounded-branching). In the sequel, we shall adopt this assumption.

Probabilistic Bisimulations. Let $\mathfrak{S} := \langle S; \{\delta_a\}_{a \in \text{ACT}} \rangle$ be a PTS. We write $s \xrightarrow{\rho} a S'$ if $\sum_{s' \in S'} \delta_a(s, s') = \rho$. A *probabilistic bisimulation* for \mathfrak{S} is an equivalence relation R over S , such that $(p, q) \in R$ implies

$$\forall a \in \text{ACT}. \forall S' \in S/R. (p \xrightarrow{\rho} a S' \Leftrightarrow q \xrightarrow{\rho} a S'). \quad (1)$$

We say that p and q are *probabilistic bisimilar* (written as $p \sim q$) if there is a probabilistic bisimulation R such that $(p, q) \in R$. We can compute probabilistic bisimulation between two PTSs $\mathfrak{S} := \langle S; \{\delta_a\}_{a \in \text{ACT}} \rangle$ and $\mathfrak{S}' := \langle S'; \{\delta'_a\}_{a \in \text{ACT}} \rangle$ by computing a probabilistic bisimulation R for the disjoint union of \mathfrak{S} and \mathfrak{S}' , which is defined as $\mathfrak{S} \sqcup \mathfrak{S}' := \langle S \sqcup S'; \{\delta''_a\}_{a \in \text{ACT}} \rangle$ where $\delta''_a(s) := \delta_a(s)$ for $s \in S$, and $\delta''_a(s) := \delta'_a(s)$ for $s \in S'$. In such case, we say R is a probabilistic bisimulation between \mathfrak{S} and \mathfrak{S}' .

3 Framework of Regular Relations

In this section we describe the framework of regular relations for specifying probabilistic infinite-state systems, properties to verify, and proofs, all in a uniform symbolic way. The framework is amenable to automata-theoretic algorithms in the spirit of *regular model checking* [3,13].

The framework of *regular relations* [8] (a.k.a. *automatic relations* [9]) uses the first-order theory of universal¹ automatic structure

$$\mathfrak{U} := \langle \Sigma^*; \preceq, \text{eqL}, \{l_a\}_{a \in \Sigma} \rangle, \quad (2)$$

where Σ is some finite alphabet, \preceq is the (non-strict) prefix-of relation, eqL is the binary equal length predicate, and l_a is a unary predicate asserting that the last letter of the word is a . The domain of the structure is the set of finite words over Σ , and for words $w, w' \in \Sigma^*$, we have $w \preceq w'$ iff there is some $w'' \in \Sigma^*$ such that $w \cdot w'' = w'$, $\text{eqL}(w, w')$ iff $|w| = |w'|$, and $l_a(w)$ iff there is some $w'' \in \Sigma^*$ such that $w = w'' \cdot a$.

Next, we discuss the expressive power of first-order formulas over the universal automatic structures, and decision procedures for satisfiability of such formulas. In Sect. 4, we shall describe: (1) how to specify a PTS as a first-order formula in \mathfrak{U} , and (2) how to specify the verification condition for probabilistic bisimulation property in this theory. In Sect. 5, we shall show that the theory is sufficiently powerful for capturing probabilistic bisimulations for interesting examples.

¹ Here, “universal” simply means that all automatic structures are first-order interpretable in this structure.

Expressiveness and Decidability. The name “regular” associated with this framework is because the set of formulas $\varphi(x_1, \dots, x_k)$ first-order definable in \mathfrak{U} coincides with *regular relations*, i.e., relations definable by synchronous automata. More precisely, we define $\llbracket \varphi \rrbracket$ as the relation which contains all tuples $(w_1, \dots, w_k) \in (\Sigma_{\perp}^*)^k$ such that $\mathfrak{U} \models \varphi(w_1, \dots, w_k)$. In addition, we define the *convolution* $w_1 \otimes \dots \otimes w_k$ of words $w_1, \dots, w_k \in \Sigma^*$ as a word w over Σ_{\perp}^k (where $\perp \notin \Sigma$) such that $w[i] = (a_1, \dots, a_k)$ with

$$a_j = \begin{cases} w_j[i] & \text{if } |w_j| \geq i, \text{ or} \\ \perp & \text{otherwise.} \end{cases}$$

In other words, w is obtained by juxtaposing w_1, \dots, w_k and padding the shorter words with \perp . For example, $010 \otimes 00 = (0, 0)(1, 0)(0, \perp)$. A k -ary relation R over Σ^* is *regular* if the set $\{w_1 \otimes \dots \otimes w_k : (w_1, \dots, w_k) \in R\}$ is a regular language over the alphabet Σ_{\perp}^k . The relationship between \mathfrak{U} and regular relations can be formally stated as follows.

Proposition 1 ([8–10]).

1. Given a formula $\varphi(\bar{x})$ over \mathfrak{U} , the relation $\llbracket \varphi \rrbracket$ is effectively regular. Conversely, given a regular relation R , we can compute a formula $\varphi(\bar{x})$ over \mathfrak{U} such that $\llbracket \varphi \rrbracket = R$.
2. The first-order theory of \mathfrak{U} is decidable.

The decidability of the first-order theory of \mathfrak{U} follows using a standard automata-theoretic algorithm (e.g. see [9, 49]).

In the sequel, we shall also use the term regular relations to denote relations definable in \mathfrak{U} . In addition, to avoid notational clutter, we shall freely use other regular relations (e.g. successor relation \prec_{succ} of the prefix \preceq , and membership in a regular language) as syntactic sugar.

We note that the first-order theory of \mathfrak{U} can also be reduced to weak monadic theory WS1S of one successor (therefore, allowing highly optimized tools like MONA [31] and Gaston [23]) by translating finite words to finite sets. The relationship between the universal automatic structure and WS1S can be made precise using the notion of *finite-set interpretations* [19, 46].

4 Probabilistic Bisimilarity Within Regular Relations

In this section, we show how the framework of regular relations can be used to encode a PTS, and the corresponding proof rules for probabilistic bisimulation.

4.1 Specifying a Probabilistic Transition System

Since we assume that all probability values specified in our systems are rational numbers, the fact that our PTS is bounded-branching implies that we can specify the probability values by natural *weights* (by multiplying the probability values by the least common multiple of the denominators). For example, if a configuration c has an action

toss that takes it to c_1 and c_2 , each with probability $1/2$, then the new system simply changes both values of $1/2$ to 1 . This is a known trick in the literature of probabilistic verification (e.g. see [1]). Therefore, we can now assume that the transition probability functions have range \mathbb{N} . *The challenge now is that our encoding of a PTS in the universal automatic structure must encode two different sorts* as words over a finite alphabet Σ : configurations and natural weights.

Now we are ready to show how to specify a PTS \mathfrak{S} in our framework. Fix a finite alphabet Σ containing at least two letters 0 and 1 . We encode the domain of \mathfrak{S} as words over Σ . In addition, a natural weight $n \in \mathbb{N}$ can be encoded in the usual way as a binary string. This motivates the following definition.

Definition 1. Let \mathfrak{S} be a PTS $\langle S; \{\delta_a\}_{a \in \text{ACT}} \rangle$. We say that \mathfrak{S} is regular if the domain S is a regular subset of Σ^* (i.e. definable by a first-order formula $\varphi(x)$ with one free variable over \mathfrak{U}), and if the graph of each function δ_a is a ternary regular relation (i.e. definable by a first-order formula $\varphi(x, y, z)$ over \mathfrak{U} , where x and y encode configurations, and z encodes a natural weight).

Definition 1 is quite general since it allows for an infinite number of different natural weights in the PTS. Note that we can make do without the second sort (of numeric weights) if we have only finitely many numeric weights n_1, \dots, n_m . This can be achieved by specifying a regular relation $R_{a,i}$ for each action label $a \in \text{ACT}$ and numeric weight n_i with $i \in \{1, \dots, m\}$.

Example 1. We show a regular encoding of a very simple PTS: a random walk on the set of natural numbers. At each position x , the system can non-deterministically choose to loop or to move. If the system chooses to loop, it will stay at the same position with probability 1. If the system chooses to move, it will move to $x + 1$ with probability $1/4$, or move to $\max(0, x - 1)$ with probability $3/4$. Normalising the probability values by multiplying by 4, we obtain the numeric weights of 4, 1, and 3 for the aforementioned transitions, respectively.

To represent the system by regular relations, we encode the positions in unary and the numeric weights in binary. The set of configurations is the regular language 1^* . The graph of the transition probability function can be described by a first-order formula $\varphi(x, y, z) := \varphi_{\text{loop}}(x, y, z) \vee \varphi_{\text{move}}(x, y, z)$ over \mathfrak{U} , where

$$\begin{aligned} \varphi_{\text{loop}}(x, y, z) &:= x \in 1^* \wedge y \in 1^* \wedge ((x = y \wedge z = 100) \vee (x \neq y \wedge z = 0)); \\ \varphi_{\text{move}}(x, y, z) &:= x \in 1^* \wedge y \in 1^* \wedge ((x \prec_{\text{succ}} y \wedge z = 1) \vee \\ &\quad (y \prec_{\text{succ}} x \wedge z = 11) \vee (x = \varepsilon \wedge y = \varepsilon \wedge z = 11) \vee \\ &\quad (\neg(x \prec_{\text{succ}} y) \wedge \neg(y \prec_{\text{succ}} x) \wedge \neg(x = \varepsilon \wedge y = \varepsilon) \wedge z = 0)). \end{aligned}$$

□

Example 2. As a second example, consider a PTS (from [25], Example 1) described by a probabilistic pushdown automaton with states $Q = \{p, q, r\}$ and stack symbols $\Gamma = \{X, X', Y, Z\}$. There is a unique action a , and the transition rules δ_a are as follows:

$$\begin{array}{llll} pX \xrightarrow{0.5} qXX & pX \xrightarrow{0.5} p & qX \xrightarrow{1} pXX & rY \xrightarrow{1} rXX \\ rX \xrightarrow{0.3} rYX & rX \xrightarrow{0.2} rYX' & rX \xrightarrow{0.5} r & \\ rX' \xrightarrow{0.4} rYX & rX' \xrightarrow{0.1} rYX' & rX' \xrightarrow{0.5} r & \end{array}$$

A configuration of the PTS is a word in $Q\Gamma^*$, consisting of a state in Q and a word over the stack symbols. A transition can be applied if the prefix of the configuration matches the left hand side of the transition rules above. We encode the PTS as follows: the set of configurations is $Q\Gamma^*$, the weights are represented in binary after normalization, and the transition relation $\varphi(x, y, z)$ encodes the transition rules in disjunction. For example, the disjunct corresponding to the rule $pX \xrightarrow{0.5} qXX$ is

$$x \in Q\Gamma^* \wedge y \in Q\Gamma^* \wedge (\exists u. x = pXu \wedge y = qXXu) \wedge z = 101.$$

Note that the PTS is bounded branching with a bound 3. \square

4.2 Proof Rules for Probabilistic Bisimulation

Fix the set ACT of action symbols and the branching bound $N \geq 1$, owing to the minimal probability assumption. Consider a two-sorted vocabulary $\sigma = \langle \{P_a\}_{a \in \text{ACT}}, R, + \rangle$, where P_a is a ternary relation (with the first two arguments over the first sort, and the third argument over the second sort of natural numbers), R is a binary relation over the first sort, and $+$ is the addition function over the second sort of natural numbers. The main result we shall show next is summarized in the following theorem:

Theorem 1. *There is a fixed first-order formula Φ over σ such that a binary relation R is a probabilistic bisimulation over a bounded-branching PTS $\mathfrak{S} = \langle S; \{\delta_a\}_{a \in \text{ACT}} \rangle$ iff $(\mathfrak{S}, R) \models \Phi$. Furthermore, when \mathfrak{S} is a regular PTS and R is a regular relation, we can compute in polynomial time a first-order formula Φ' over \mathfrak{U} such that R is a probabilistic bisimulation over \mathfrak{S} iff $\mathfrak{U} \models \Phi'$.*

This theorem implies the following result:

Theorem 2. *Given a regular relation $E \subseteq \Sigma^* \times \Sigma^*$ and a bounded-branching regular PTS $\mathfrak{S} = \langle S; \{\delta_a\}_{a \in \text{ACT}} \rangle$, there exists an algorithm that either finds $(u, v) \in E$ which are not probabilistically bisimilar or finds a regular probabilistic bisimulation relation R over \mathfrak{S} such that $E \subseteq R$ if one exists. The algorithm does not terminate iff E is contained in some probabilistic bisimulation relation but every probabilistic bisimulation R containing E is not regular.*

Note that when verifying parameterized systems we are typically interested in checking bisimilarity over a set of pairs (instead of just one pair) of configurations, and hence E in the above statement.

Proof of Theorem 2. To prove this, we provide two semi-algorithms, one for checking the existence of R and the other for showing that a pair $(v, w) \in E$ is a witness for non-bisimilarity.

By Theorem 1, we can enumerate all possible candidate regular relation R and effectively check that R is a probabilistic bisimulation over \mathfrak{S} . The condition that $E \subseteq R$ is a first-order property, and so can be checked effectively.

To show non-bisimilarity is recursively enumerable, observe that if we fix $(v, w) \in E$ and a number d , then the restrictions \mathfrak{S}_v and \mathfrak{S}_w to configurations that are of distance

at most d away from v and w (respectively) are finite PTS. Therefore, we can devise a semi-algorithm which enumerates all $(v, w) \in E$, and all probabilistic modal logic (PML) formulas [34] F over ACT containing only rational numbers (i.e. a formula of the form $\langle a \rangle_\mu F'$, where $\mu \in [0, 1]$ is a rational number, which is sufficient because we assume only rational numbers in the PTS). We need to check that $\mathfrak{S}_v, v \models F$, but $\mathfrak{S}_w, w \not\models F$. Model checking PML formulas over finite systems is decidable (in fact, the logic is subsumed by Probabilistic CTL [7]), which makes our check effective. \square

4.3 Proof of Theorem 1

In the rest of the section, we shall give a proof of Theorem 1. Given a binary relation $R \subseteq S \times S$, we can write a first-order formula $F_{eq}(R)$ for checking that R is an equivalence relation:

$$\forall s, t, u \in S. R(s, s) \wedge (R(s, t) \Rightarrow R(t, s)) \wedge ((R(s, t) \wedge R(t, u) \Rightarrow R(s, u)).$$

We shall next define a formula $\varphi_a(p, q)$ for each $a \in \text{ACT}$, such that R is a probabilistic bisimulation for $\mathfrak{S} = \langle S; \{\delta_a\}_{a \in \text{ACT}} \rangle$ iff $(\mathfrak{S}, R) \models \Phi(R)$, where

$$\Phi(R) := F_{eq}(R) \wedge \forall p, q \in S. R(p, q) \Rightarrow \bigwedge_{a \in \text{ACT}} (\psi_a(p) \wedge \psi_a(q)) \vee \varphi_a(p, q). \quad (3)$$

The formula $\psi_a(s) := \forall s' \in S. \delta_a(s, s') = 0$ states that configuration s cannot move to any configuration through action a .

Before we describe $\varphi_a(p, q)$, we provide some intuition and define some intermediate macros. Fix configurations p and q . Informally, $\varphi_a(p, q)$ will first guess a set of configurations u_1, \dots, u_N containing the successors of p on action a , and a set of configurations v_1, \dots, v_N containing the successors of q on action a . Second, it will guess labellings $\alpha_1, \dots, \alpha_N$ and β_1, \dots, β_N which correspond to partitionings of the configurations u_1, \dots, u_N and v_1, \dots, v_N , respectively. The intuition is that the α 's and β 's “name” the partitions: if $\alpha_i = \alpha_j$ (resp. $\beta_i = \beta_j$), then u_i and u_j (resp. v_i and v_j) are guessed to be in the same partition. The formula then checks that the guessed partitioning is compatible with the equivalence relation R (i.e. if the labelling claims u_i and u_j are in the same partition, then indeed $R(u_i, u_j)$ holds), and that the probability masses of the partitions assigned by configurations p and q satisfy the constraint given in (1).

For the first part, we define a formula

$$\begin{aligned} \text{succ}_a(w; u_1, \dots, u_N) := & \left(\bigwedge_{1 \leq i < j \leq N} u_i \neq u_j \right) \wedge \\ & \left(\forall u \in S. \delta_a(w, u) \neq 0 \Rightarrow \bigvee_{1 \leq i \leq N} u = u_i \right), \end{aligned}$$

stating that the successors of configuration w on action a are among the N distinct configurations u_1, \dots, u_N . Note that a configuration may have fewer than N successors. In this case, we can set the rest of the variables to arbitrary distinct configurations.

For the second part, we shall check that R is compatible with the guessed partitions, and that configurations p and q assign the same probability mass to the same partition.

Let k_1, \dots, k_n be a labelling for configurations s_1, \dots, s_n . To check that the partitioning induced by the labelling is compatible with R , we need to express the condition that $k_i = k_j$ if and only if $R(s_i, s_j)$ holds. To this end, we define a formula

$$\text{compat}_R(s_1, \dots, s_n; k_1, \dots, k_n) := \bigwedge_{1 \leq i < j \leq n} (R(s_i, s_j) \Leftrightarrow k_i = k_j).$$

Now, we are ready to define $\varphi_a(p, q)$:

$$\begin{aligned} \varphi_a(p, q) := & \exists u_1, \dots, u_N, v_1, \dots, v_N \in S. \exists \alpha_1, \dots, \alpha_N, \beta_1, \dots, \beta_N \in \mathbb{N}. \\ & \text{succ}_a(p; u_1, \dots, u_N) \wedge \text{succ}_a(q; v_1, \dots, v_N) \wedge \\ & \text{compat}_R(u_1, \dots, u_N, v_1, \dots, v_N; \alpha_1, \dots, \alpha_N, \beta_1, \dots, \beta_N) \wedge \\ & \forall k \in \mathbb{N}. \left(\sum_{i: \alpha_i=k} \delta_a(p, u_i) = \sum_{i: \beta_i=k} \delta_a(q, v_i) \right). \end{aligned} \quad (4)$$

With this definition, $\varphi_a(p, q)$ holds if and only if $p \xrightarrow{\rho} a S' \Leftrightarrow q \xrightarrow{\rho} a S'$ holds for any $\rho \geq 0$ and equivalence class $S' \in S/R$.

Example 3. Consider the PTS from Example 2. The configurations pXZ and rX are probabilistic bisimilar. This can be seen using a probabilistic bisimulation relation with equivalence classes $\{pX^k Z\} \cup \{rw : w \in \{X, X'\}^k\}$ for all $k \geq 0$ and $\{qX^{k+1} Z\} \cup \{rYw : w \in \{X, X'\}^k\}$ for all $k \geq 1$. The probabilistic bisimulation relation is definable as the symmetric closure of a regular relation R , where $(w_1, w_2) \in R$ iff

$$\begin{aligned} & (w_1 = w_2) \vee \\ & (w_1 \in pX^* Z \wedge w_2 \in r(X + X')^* \perp \wedge |w_1| = |w_2|) \vee \\ & (w_1 \in r(X + X')^* \wedge w_2 \in r(X + X')^* \wedge |w_1| = |w_2|) \vee \\ & (w_1 \in qX^* Z \wedge w_2 \in rY(X + X')^* \perp \wedge |w_1| = |w_2|) \vee \\ & (w_1 \in rY(X + X')^* \wedge w_2 \in rY(X + X')^* \wedge |w_1| = |w_2|). \end{aligned}$$

For this example, the formula (3) simplifies to $F_{eq}(R) \wedge \forall s, t \in S. \varphi_a(p, q)$ for the unique action a . This formula defines a condition that checks the bisimulation relation for all states symbolically. To see the formula in action, fix configurations pXZ and rX which are probabilistic bisimilar. In the PTS, pXZ has two successors, $qXXZ$ and pZ , each with probability 0.5, and rX has three successors, rYX with probability 0.3, rYX' with probability 0.2, and r with probability 0.5. In the formula for $\varphi_a(p, q)$, we can set the successors u_i of pXZ and the successors v_j of rX as above (the third “successor” u_3 is set to an arbitrary configuration not reachable from pXZ), and set $\alpha_1 = 1, \alpha_2 = 2, \beta_1 = \beta_2 = 1$, and $\beta_3 = 2$, corresponding to the equivalence classes of the bisimulation relation. One can check that the probability masses to these classes are the same.

We remark that the first-order theory of \mathfrak{U} is sufficient to encode any probabilistic pushdown automaton, not just this example. \square

We proceed to show that if R and δ_a are first-order definable over \mathfrak{U} then so are ψ_a and φ_a . Suppose that δ_a is encoded using the ternary relation $\delta_a(x, y, z)$, as stated in the previous section. (We shall re-use the symbol δ here to avoid a clash of names).

We define $\psi_a(s) := \forall s' \in S. \forall z \in \mathbb{N}. \delta_a(s, s', z) \Leftrightarrow z = 0$. To define φ_a , the key point is to express the sum of transition probabilities in the logic. We use the fact that addition of integers in binary encoding is regular (see e.g. [9]), and write a formula that performs iterated addition. Formally, for each $a \in \text{ACT}$ we define a formula χ_a such that

$$\chi_a(u; u_1, \dots, u_N; \alpha_1, \dots, \alpha_N; k; z) :=$$

$$\exists z_1, \dots, z_{N+1} \in \mathbb{N}. z_1 = 0 \wedge z_{N+1} = z \wedge \bigwedge_{1 \leq i \leq N} \chi'_a(u, u_i, \alpha_i, k, z_i, z_{i+1}),$$

where

$$\chi'_a(u, u', \kappa, k, x, y) := (\kappa = k \wedge \exists z. \delta_a(u, u', z) \wedge y = x + z) \vee (\kappa \neq k \wedge y = x)$$

performs a single addition—we use the fact that addition “ $y = x + z$ ” in binary is encodable as a regular relation—and z_1, \dots, z_{N+1} store the intermediate sums. Hence, given $k \in \mathbb{N}$, $u_1, \dots, u_N, v_1, \dots, v_N \in S$, and $\alpha_1, \dots, \alpha_N, \beta_1, \dots, \beta_N \in \mathbb{N}$,

$$\sum_{i: \alpha_i=k} \delta_a(p, u_i) = \sum_{i: \beta_i=k} \delta_a(q, v_i)$$

if and only if

$$\exists z \in \mathbb{N}. \chi_a(p; u_1, \dots, u_N; \alpha_1, \dots, \alpha_N; k; z) \wedge \chi_a(q; v_1, \dots, v_N; \beta_1, \dots, \beta_N; k; z).$$

It follows that $\varphi_a(p, q)$ defined in (4) can be encoded in the first-order theory of \mathfrak{U} .

Remark. Note that checking the validity of a given presentation of a regular PTS is algorithmic. To see this, suppose we are given a set of formulae $\{\delta_a(x, y, z)\}_{a \in \text{ACT}}$ that is claimed to encode the probabilistic transition functions of a PTS with a branching bound N . Fix a formula δ_a . First, we need to check that for all $x \in S$, there are at most N distinct y 's such that $\delta_a(x, y, z)$ satisfies $z \neq 0$. Second, we need to check that $\llbracket \delta_a \rrbracket$ is a function, i.e., $\forall x, y. \exists! z. \delta_a(x, y, z)$, where $\exists! z. \varphi(\bar{x}, z)$ is a shorthand for the formula asserting there exists precisely one z such that $\varphi(\bar{x}, z)$ is true. Third, we need to check that $\llbracket \delta_a \rrbracket$ encodes a mapping $S \rightarrow \{\bar{0}\} \cup \mathcal{D}_S$. The first two requirements are easily seen to be expressible as a first-order formula and hence is algorithmic over \mathfrak{U} . The third requirement amounts to checking the assertion that there exists $w_a \in \mathbb{N}$ satisfying

$$\begin{aligned} \forall x \in S. (\forall y \in S. \forall z \in \mathbb{N}. \delta_a(x, y, z) \Leftrightarrow z = 0) \vee \\ (\exists y_1, \dots, y_N \in S. \exists z_1, \dots, z_N \in \mathbb{N}. \\ \text{succ}_a(x; y_1, \dots, y_N) \wedge \bigwedge_{1 \leq i \leq N} \delta_a(x, y_i, z_i) \wedge \sum_{1 \leq i \leq N} z_i = w_a), \end{aligned}$$

which is a first-order formula and is algorithmic over \mathfrak{U} by the fact that summation of a fixed number of weights is regular (as shown earlier in this section). Finally, since all of the w_a 's are expected to be the same common multiple of the denominators of the transition probabilities, we need to check that there is $w \in \mathbb{N}$ such that $w_a = w$ for all $a \in \text{ACT}$. This is again algorithmic as we can pinpoint the exact value of each w_a by enumeration.

5 Application to Anonymity Verification

In this section, we show how to verify the anonymity property of cryptographic protocols via computation of probabilistic bisimulations. We shall first formalize the connection between the concepts of anonymity and probabilistic bisimulation. We then introduce a verification framework and apply it to verify the anonymity property of the dining cryptographers protocol [16] and the grades protocol [29].

A (*discrete time*) *Markov chain* (a.k.a. *DTMC*) is a structure $\mathfrak{M} := \langle S; \delta; L \rangle$ where S is a set of configurations, $\delta : S \rightarrow \mathcal{D}_S$ is a family of probability distributions, and $L : S \rightarrow \text{ACT}$ is a labelling of the states. We shall use $\delta(s, s')$ to denote $\delta(s)(s')$, the transition probability from s to s' . A sequence $s_0 \dots s_n \in S^*$ is called a *path* of \mathfrak{M} if $\delta(s_i, s_{i+1}) > 0$ for $i \in \{0, \dots, n-1\}$. The probability distribution induced by the paths in a DTMC can be defined using a standard cylinder construction (see e.g. [33]) as follows. Given a finite path $\pi := s_0 \dots s_n \in S^*$, we set Run_π to be a *basic cylinder*, which is the set of all finite/infinite paths with π as a prefix. We associate this cylinder with probability $\Pr^{s_0}(\text{Run}_\pi) = \prod_{i=0}^{n-1} \delta(s_i, s_{i+1})$. This gives rise to a unique probability measure for the σ -algebra over the set of all paths from s_0 .

Given a PTS $\mathfrak{G} := \langle S; \{\delta_a\}_{a \in \text{ACT}} \rangle$, an *adversary* $f : S^* \rightarrow \text{ACT}$ resolves the non-determinacy of \mathfrak{G} and induces a DTMC $\mathfrak{G}_f := \langle S'; \delta'; L' \rangle$. Here $S' := S^* \cup \{\$\}$ contains all finite paths of \mathfrak{G} plus a “sink state” $\$$ such that $\delta'(\pi) := I_\$$ ² if and only if either $\pi = \$$, or $\delta_{f(\pi)}$ is the zero function. We define $\delta'(\pi) := \delta_{f(\pi)}$ otherwise. The labelling of \mathfrak{G}_f is defined as $L'(\$) := \perp$ and $L'(\pi) := f(\pi)$ for $\pi \in S^*$.

Given a DTMC $\langle S; \delta; L \rangle$, the *trace* of a path $\pi := s_0 \dots s_n \in S^*$ is defined as $\tau(\pi) := L(s_0) \dots L(s_n)$. A *trace event* \mathcal{T} is a set of finite traces; the probability of \mathcal{T} with respect to a configuration s is specified with $\Pr^s(\mathcal{T}) := \Pr^s(\bigcup\{\text{Run}_\pi : \tau(\pi) \in \mathcal{T}, \pi \text{ starts from } s\})$.

Now we are ready to define the concept of anonymity. Fix $\mathfrak{G} := \langle S; \{\delta_a\}_{a \in \text{ACT}} \rangle$ and a set $\mathcal{I} \subseteq S$ of initial configurations. We say \mathfrak{G} is *anonymous to an adversary* f if for all $s \in \mathcal{I}$ and trace event \mathcal{T} , the value of $\Pr^s(\mathcal{T})$ in \mathfrak{G}_f is solely determined by \mathcal{T} . Intuitively, this means that the adversary cannot obtain any information about a specific initial configuration by experimenting on the system and observing the traces.

We shall only consider external adversaries in this paper. An adversary $f : S^* \rightarrow \text{ACT}$ is *external* if $f(s_0 \dots s_n) = f(s'_0 \dots s'_n)$ when $L(s_i) = L(s'_i)$ for $i \in \{0, \dots, n\}$. That is, an external adversary takes action solely based on the trace she has observed so far. We call a PTS *anonymous* if it is anonymous to any external adversary. The following result establishes a connection between the anonymity property and probabilistic bisimulations.

Proposition 2. *Let $\mathfrak{G} := \langle S; \{\delta_a\}_{a \in \text{ACT}} \rangle$ be a PTS and f be an external adversary for \mathfrak{G} . Then for all $u, v \in S$ such that $u \sim v$, $\Pr^u(\mathcal{T}) = \Pr^v(\mathcal{T})$ holds for any trace event \mathcal{T} in \mathfrak{G}_f . That is, configurations u and v induce the same trace distribution in \mathfrak{G}_f .*

Based on Proposition 2, we propose a framework to verify the anonymity property of \mathfrak{G} as follows. We first specify a “reference system” $\mathfrak{G}' := \langle S; \{\delta'_a\}_{a \in \text{ACT}} \rangle$ that has

² Recall that I_s denotes the point distribution at s , namely $I_s(s) = 1$.

the same initial configurations and actions as those of \mathfrak{S} , except that the trace distribution of \mathfrak{S}'_f is independent of specific initial configurations for any adversary f . We then try to find a bisimulation relation R between \mathfrak{S} and the reference system \mathfrak{S}' satisfying $R \supseteq \{(s, s') \in \mathcal{I} \times \mathcal{I}' : s = s'\}$. When such a relation R is found, we can conclude that the trace distribution of \mathfrak{S}_f is also independent of the initial configurations for any adversary f , and hence prove the anonymity property of \mathfrak{S} .

The Dining Cryptographers Protocol. Dining cryptographers protocol [16] is a multi-party computation algorithm aiming to securely compute the XOR of the secret bits held by the participants. More precisely, consider a ring of $n \geq 3$ participants p_0, \dots, p_{n-1} such that each participant p_i holds a secret bit x_i . To compute $x_0 \oplus \dots \oplus x_{n-1}$ without revealing information about the values of x_0, \dots, x_{n-1} , the participants carry out a two-stage computation as follows: (i) Each two adjacent participants p_i, p_{i+1} compute a random bit b_i that is accessible only to them; (ii) Each participant p_i announces the value $a_i := x_i \oplus b_i \oplus b_{i-1}$ ³ to the other participants. Hence, every participant p_i can observe the values of x_i, b_i, b_{i-1} and a_0, \dots, a_{n-1} . It turns out that $a_0 \oplus \dots \oplus a_{n-1} = x_0 \oplus \dots \oplus x_{n-1}$, so all participants are able to compute the XOR of the secret bits after executing the protocol. Furthermore, the anonymity property of the protocol assures that any individual participant p_i cannot infer the values of the other secret bits from the information she has observed during the execution of the protocol.

We model the protocol as a length-preserving regular PTS. The configurations of a ring of n participants are encoded as words of size n . The initial configurations are words $w \in \{0, 1\}^*$ such that $w[i]$ represents x_i for $i \in \{0, \dots, |w| - 1\}$. The transition relation consists of six transitions: observer non-deterministically tossing head (via action head), observer non-deterministically tossing tail (via action tail), non-observer tossing head with probability 0.5 (via action toss), non-observer tossing tail with probability 0.5 (via action toss), participant announcing zero (via action zero), and participant announcing one (via action one). The outcomes of the tosses by the observer are visible (i.e. as actions head and tail), while the outcomes of the tosses by the other participants are hidden (i.e. as action toss). Each maximal trace from an initial configuration of size n consists of n successive tossing actions, followed by n successive announcing actions. Starting from an initial configuration w and for $i \in \{0, \dots, n - 1\}$, the i -th toss action updates the value of $w[j]$ to $w[j] \oplus b_i$ for $j \in \{i, i + 1\}$, where $b_i = 1$ if a head is tossed and $b_i = 0$ otherwise. Any configuration v reached after n tosses would satisfy $v[i] = x_i \oplus b_i \oplus b_{i-1}$ for $i \in \{0, \dots, n - 1\}$. The PTS then “prints out” the configuration by going through n announcement transitions via actions a_0, \dots, a_{n-1} , such that a_i is one if $v[i] = 1$ and a_i is zero if $v[i] = 0$.

We consider the case where the first participant of the protocol is the observer. The maximal traces of the PTS in this case are in form of $t \cdot t'$, where $|t| = |t'|$, $t \in \{\text{head}, \text{tail}\}$ toss* $\{\text{head}, \text{tail}\}$, and $t' \in \{\text{zero}, \text{one}\}^*$. For example, head toss tail one zero zero is a maximal trace starting from initial configuration 010. To prove anonymity, we define a reference system such that the initial configurations and the actions are the same as those of the original PTS, except that the announcements a_0, \dots, a_{n-1}

³ All arithmetical operations on the subscripts are performed modulo n to take the ring structure into account.

encoded in the maximal traces from an initial configuration w are uniformly distributed over $\{(a_0, \dots, a_{n-1}) : a_0 \oplus \dots \oplus a_{n-1} = w[0] \oplus \dots \oplus w[n-1], a_0 = w[0] \oplus b_0 \oplus b_{n-1}\}$.⁴ In this way, the distribution of the announcements is independent of the initial configuration once the values of $x_0 \oplus \dots \oplus x_{n-1}$, x_0 , b_0 , and b_{n-1} (i.e. the information revealed to the first participant) are fixed. We then compute a probabilistic bisimulation between the original system and the reference system, establishing the anonymity property that the first participant cannot infer the secret bits of the other participants from the information she observes.

A generalized Dining Cryptographers Protocol. We have also considered a generalized dining cryptographers protocol where the secret messages x_0, \dots, x_{n-1} of the n participants are bit-vectors of the same size. Note that the set of the initial configurations is not regular when the size of the secret messages is parameterized. To construct a regular model, we allow a configuration to encode secret messages of different sizes, and devise the transition system such that an initial configuration w can finish the protocol (i.e. can have a trace containing all of the announcements a_0, \dots, a_{n-1}) if and only if the messages encoded in w have same size. The resulting PTS is a regular system; it over-approximates the PTS of the generalized dining cryptographers protocol in the sense that the anonymity property of the former implies that of the latter.

The Grades Protocol. The grades protocol [29] is a multi-party computation algorithm aiming to securely compute the sum of the secrets held by the participants. The setting of the protocol is pretty similar to that of the dining cryptographers: given $n \geq 3$ and $g \geq 2$, we have a ring of n participants p_0, \dots, p_{n-1} where each participant p_i holds a secret $x_i \in \{0, \dots, g-1\}$. Note that both g and n are parameterized in this protocol. The goal of the participants is to compute the sum $x_0 + \dots + x_{n-1}$ without revealing information about the individual secrets. Define $M := (g-1) \cdot n + 1$. The protocol consists of two steps: (i) Each two adjacent participants p_i, p_{i+1} compute a random number $y_i \in \{0, \dots, M-1\}$; (ii) Each participant p_i announces $a_i := (x_i + y_i - y_{i-1}) \bmod M$ to the other participants. After executing the protocol, the participants compute $a := a_0 + \dots + a_{n-1} \bmod M$. Because of the ring structure, the y_i 's will be cancelled out in the sum. Thus the value of a will equal to the sum of all secrets. The anonymity property of the protocol asserts that no participant can infer the secrets held by the other participants from the information she has observed.

We consider a variant of the grades protocol where M can be any power of two greater than $(g-1) \cdot n$. Observe that the same anonymity and correctness property of the original protocol also holds for this variant. To verify the anonymity property, we model an over-approximation of the protocol where the secrets are allowed to range over $\{0, \dots, M-1\}$. This model is similar to the one we have constructed for the generalized dining cryptographers protocol except that, e.g., the XOR operations are now replaced with bitwise additions and negations. A reference system is specified such that the announcements a_1, \dots, a_{n-1} observed by the first participant p_0 are uniformly distributed over the values satisfying $a_0 + \dots + a_{n-1} \bmod M = x_0 + \dots + x_{n-1} \bmod M$.

⁴ Such a distribution can be obtained by (i) choose $a_1, \dots, a_{n-2} \in \{0, 1\}$ uniformly at random; (ii) set $a_0 = w[0] \oplus b_0 \oplus b_{n-1}$; (iii) set $a_{n-1} = a_0 \oplus \dots \oplus a_{n-2} \oplus w[0] \oplus \dots \oplus w[n-1]$.

Algorithm 1. Equivalence check for L^*

Input: Candidate automaton \mathcal{H} over $\Sigma \times \Sigma$, PTS \mathfrak{S} , and relation $E \subseteq (\Sigma \times \Sigma)^*$.

Result: $NoSolution(v, w)$ if there is no bisimulation R with $E \subseteq R$.
 $PositiveCEX(v, w)$ if \mathcal{H} should accept (v, w) , but does not;
 $NegativeCEX(v, w)$ if \mathcal{H} accepts (v, w) , but should not;
 $Correct$ if \mathcal{H} is a correct bisimulation for PTS \mathfrak{S} and $E \subseteq \mathcal{L}(\mathcal{H})$;

```

1 Check whether  $E \subseteq \mathcal{L}(\mathcal{H})$ , and whether  $\mathfrak{S} \models \Phi(\mathcal{L}(\mathcal{H}))$  using the  $\Phi$  from (3);
2 if there is a counterexample of minimal length  $n$  then
3   Compute the greatest bisimulation  $\bar{R}_n$  restricted to configurations of length  $n$ ;
4   if there is  $(v \otimes w) \in E \setminus \bar{R}_n$  with  $|v| = |w| = n$  then
5     Output  $NoSolution(v, w)$  and abort;
6   else if there is  $(v \otimes w) \in \mathcal{L}(\mathcal{H}) \setminus \bar{R}_n$  with  $|v| = |w| = n$  then
7     return  $NegativeCEX(v, w)$ ;
8   else if there is  $(v \otimes w) \in \bar{R}_n \setminus \mathcal{L}(\mathcal{H})$  then
9     return  $PositiveCEX(v, w)$ ;
10 else
11   return  $Correct$ ;
```

By computing a probabilistic bisimulation between the original system and the reference system, we establish the anonymity property that the grades protocol is anonymous whenever M is chosen as a power of two with $M \geq (g - 1) \cdot n + 1$.

6 Learning Probabilistic Bisimulations

We propose an automata learning method to automatically compute regular probabilistic bisimulations R , focusing on the case of *length-preserving* PTSs, which covers all examples given in the previous section. The approach uses active automata learning, for instance Angluin's L^* method [5] or refinements of it, to compute R . This approach is inspired by previous work on using active automata learning for invariant inference [18, 54]. Our procedure assumes (i) as input a bounded-branching PTS $\mathfrak{S} = \langle S; \{\delta_a\}_{a \in ACT} \rangle$, as well as a length-preserving regular relation $E \subseteq (\Sigma \times \Sigma)^*$ supposed to be covered by R ; (ii) an effective way to check the correctness of R , i.e., a decision procedure in the sense of Theorem 1; and (iii) a procedure to compute the greatest probabilistic bisimulation $\bar{R}_n \subseteq (\Sigma \times \Sigma)^n$ for \mathfrak{S} restricted to configurations of any length $n \in \mathbb{N}$. The last assumption can easily be satisfied for length-preserving PTSs. Indeed, such systems, restricted to configurations of length n , are finite-state, so that efficient existing methods [6, 17, 20, 52] apply. A solution R is presented as a deterministic letter-to-letter transducer, i.e., as a deterministic finite-state automaton over the alphabet $\Sigma \times \Sigma$.

Since L^* -style learning requires the taught language to be uniquely defined, our approach attempts to learn a representation of the greatest *length-preserving* probabilistic bisimulation relation $\bar{R} \subseteq (\Sigma \times \Sigma)^*$, which is the unique bisimulation relation formed by the union of all length-preserving probabilistic bisimulations of \mathfrak{S} , i.e., $\bar{R} = \bigcup_{n \geq 1} \bar{R}_n$. Because \bar{R} is not in general computable, the learning process might

diverge and fail to produce any probabilistic bisimulation. It can also happen that learning terminates, but yields a probabilistic bisimulation relation strictly smaller than \bar{R} .

The L^* method requires a teacher that is able to answer two kinds of queries:

- **membership queries**, i.e., whether a pair (v, w) of words should be accepted by the automaton to be learned. Since our learner tries to learn the greatest bisimulation, the teacher can answer this query by checking whether the configurations v, w are bisimilar; this is done by computing the greatest bisimulation $\bar{R}_{|v|}$ restricted to configurations of any length $|v| = |w|$, and checking whether or not $(v, w) \in \bar{R}_{|v|}$.
- **equivalence queries**, i.e., whether a candidate automaton \mathcal{H} is the correct language to be learned. Such queries can essentially be answered by checking whether the language $\mathcal{L}(\mathcal{H})$ satisfies the formula $\Phi(R)$ from (3). The complete algorithm for answering equivalence queries is given in Algorithm 1. The algorithm first attempts to find a shortest counterexample to the proof rule. If a counterexample of length n is found, then the difference set $\mathcal{L}(\mathcal{H}) \Delta \bar{R}_n$ must contain at least one pair of length n . Any of such pairs is a valid counterexample for automata learning since the learner tries to learn the greatest bisimulation. The teacher thus reports one such pair to be a positive or negative counterexample according to its membership in \bar{R}_n .

Properties of the Learning Algorithm. The learning procedure terminates when the teacher outputs *NoSolution* or returns *Correct* for an equivalence query. In the former case, the teacher explicitly provides a pair of non-bisimilar configurations in E . In the latter case, the procedure computes an automaton \mathcal{H} such that $E \subseteq \mathcal{L}(\mathcal{H})$ and $\mathcal{L}(\mathcal{H})$ is a correct probabilistic bisimulation (as it satisfies the proof rule based on Theorem 1), though not necessarily the greatest one. Since all counterexamples reported by the teacher are contained in $\mathcal{L}(\mathcal{H}) \Delta \bar{R}$, the learning procedure is guaranteed to terminate for PTSs where the greatest probabilistic bisimulation \bar{R} is regular.

Optimization with Inductive Invariants. There is a natural way to optimize the learning procedure by only considering a *regular* inductive invariant Inv such that Inv contains the set of reachable configurations and $E \subseteq Inv \times Inv$. The optimization is done by simply replacing the greatest finite-length bisimulations \bar{R}_i in Algorithm 1, and when answering membership queries, with the greatest bisimulation $\bar{R}_i^I = \bar{R}_i \cap Inv$ on the inductive invariant. Since \bar{R}_i^I can be a lot smaller than \bar{R}_i , this can lead to significant speed-ups. Note that a bisimulation R' on Inv can be extended to a bisimulation R on all configurations by setting $R = R' \cup \{(v, v) : v \notin Inv\}$. The inductive invariant Inv may be manually specified, or automatically generated using techniques like in [18, 54].

Experimental Results and Conclusion. We have implemented a prototype in Scala to test our learning method. Given a PTS specified over \mathfrak{U} , our tool first translates it to WS1S formulas and obtains finite automata for these formulas using the Mona tool [30]. Our prototype then applies the L^* learning procedure as described in this section, including the optimization to consider only the configurations of valid format. When answering an equivalence query, our tool invokes Mona to verify candidate automata and obtain counterexamples (line 1–2 of Algorithm 1). We use the prototype tool to prove the anonymity property of the three protocols described in Sect. 5. The proofs

Table 1. Experimental results. For each case study, we list the size of the final proof produced by our tool, the time taken by Mona to verify the candidate automata, the time taken by our tool to compute the fixed-length bisimulations, and the total computation time of the learning procedure. Experiments are run on a Windows laptop with 2.4 GHz Intel i5 processor and 2 GB memory limit.

Case study	#states	#trans	Mona	Bisim	Total
Dining cryptographers, single-bit	13	832	2 s	2 s	6 s
Dining cryptographers, multi-bit	16	1024	3 s	24 s	28 s
The grades protocol	25	1600	5 s	28 s	35 s

generated by our tool are finite-state automata encoding the desired probabilistic bisimulation relations. The experimental results are summarized in Table 1.

References

1. Abdulla, P.A., Henda, N.B., Mayr, R.: Decisive Markov chains. *Log. Methods Comput. Sci.* **3**(4), 1–32 (2007)
2. Abdulla, P.A., Jonsson, B., Nilsson, M., d’Orso, J., Saksena, M.: Regular model checking for LTL(MSO). *STTT* **14**(2), 223–241 (2012)
3. Abdulla, P.A., Jonsson, B., Nilsson, M., Saksena, M.: A survey of regular model checking. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 35–48. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-28644-8_3
4. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification - The KeY Book - From Theory to Practice. LNCS, vol. 10001. Springer, Cham (2016)
5. Angluin, D.: Learning regular sets from queries and counterexamples. *Inf. Comput.* **75**(2), 87–106 (1987)
6. Baier, C.: Polynomial time algorithms for testing probabilistic bisimulation and simulation. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 50–61. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61474-5_57
7. Baier, C., Katoen, J.-P.: Principles of Model Checking. MIT Press, Cambridge (2008)
8. Benedikt, M., Libkin, L., Schwentick, T., Segoufin, L.: Definable relations and first-order query languages over strings. *J. ACM* **50**(5), 694–751 (2003)
9. Blumensath, A.: Automatic structures. Diploma thesis, RWTH-Aachen (1999)
10. Blumensath, A., Grädel, E.: Finite presentations of infinite structures: automata and interpretations. *Theory Comput. Syst.* **37**(6), 641–674 (2004)
11. Boigelot, B., Legay, A., Wolper, P.: Iterating transducers in the large (extended abstract). In: Hunt, W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 223–235. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45069-6_24
12. Bonchi, F., Pouy, D.: Checking NFA equivalence with bisimulations up to congruence. In: The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2013, Rome, Italy 23–25 January 2013, pp. 457–468 (2013)
13. Bouajjani, A., Jonsson, B., Nilsson, M., Touili, T.: Regular model checking. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 403–418. Springer, Heidelberg (2000). https://doi.org/10.1007/10722167_31

14. Bradley, A.R., Manna, Z.: *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer, Heidelberg (1998)
15. Chatzikokolakis, K., Norman, G., Parker, D.: Bisimulation for demonic schedulers. In: de Alfaro, L. (ed.) FoSSaCS 2009. LNCS, vol. 5504, pp. 318–332. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00596-1_23
16. Chaum, D.: The dining cryptographers problem: unconditional sender and recipient untraceability. *J. Cryptol.* **1**(1), 65–75 (1988)
17. Chen, D., van Breugel, F., Worrell, J.: On the complexity of computing probabilistic bisimilarity. In: Birkedal, L. (ed.) FoSSaCS 2012. LNCS, vol. 7213, pp. 437–451. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28729-9_29
18. Chen, Y., Hong, C., Lin, A.W., Rümmer, P.: Learning to prove safety over parameterised concurrent systems. In: 2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, 2–6 October 2017, pp. 76–83 (2017)
19. Colcombet, T., Löding, C.: Transforming structures by set interpretations. *Log. Methods Comput. Sci.* **3**(2), 1–36 (2007)
20. Derisavi, S., Hermanns, H., Sanders, W.H.: Optimal state-space lumping in Markov chains. *Inf. Process. Lett.* **87**(6), 309–315 (2003)
21. Esparza, J., Etessami, K.: Verifying probabilistic procedural programs. In: Lodaya, K., Mahajan, M. (eds.) FSTTCS 2004. LNCS, vol. 3328, pp. 16–31. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30538-5_2
22. Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.Y.: *Reasoning About Knowledge*. MIT Press, Cambridge (2003)
23. Fiedor, T., Holík, L., Janků, P., Lengál, O., Vojnar, T.: Lazy automata techniques for WS1S. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10205, pp. 407–425. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_24
24. Flanagan, C., Leino, K., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: PLDI 02: Programming Language Design and Implementation, pp. 234–245. ACM (2002)
25. Forejt, V., Jancar, P., Kiefer, S., Worrell, J.: Bisimilarity of probabilistic pushdown automata. In: IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2012, Hyderabad, India, 15–17 December 2012, pp. 448–460 (2012)
26. Garg, P., Neider, D., Madhusudan, P., Roth, D.: Learning invariants using decision trees and implication counterexamples. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, 20–22 January 2016, pp. 499–512 (2016)
27. Habermehl, P., Vojnar, T.: Regular model checking using inference of regular languages. *Electr. Notes Theor. Comput. Sci.* **138**(3), 21–36 (2005)
28. Hinton, A., Kwiatkowska, M., Norman, G., Parker, D.: PRISM: a tool for automatic verification of probabilistic systems. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 441–444. Springer, Heidelberg (2006). https://doi.org/10.1007/11691372_29
29. Kiefer, S., Murawski, A.S., Ouaknine, J., Wachter, B., Worrell, J.: APEX: an analyzer for open probabilistic programs. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 693–698. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31424-7_51
30. Klarlund, N., Møller, A.: Mona version 1.4: User manual. BRICS, Department of Computer Science, University of Aarhus Denmark (2001)
31. Klarlund, N., Møller, A., Schwartzbach, M.I.: MONA implementation secrets. *Int. J. Found. Comput. Sci.* **13**(4), 571–586 (2002). World Scientific Publishing Company. Earlier version in Proc. 5th International Conference on Implementation and Application of Automata (CIAA) 2000, Springer-Verlag LNCS vol. 2088

32. Kundu, S., Tatlock, Z., Lerner, S.: Proving optimizations correct using parameterized program equivalence. In: Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, pp. 327–337. ACM, New York (2009)
33. Kwiatkowska, M.Z.: Model checking for probability and time: from theory to practice. In: 18th IEEE Symposium on Logic in Computer Science (LICS 2003), Ottawa, Canada, 22–25 June 2003, Proceedings, p. 351 (2003)
34. Larsen, K.G., Skou, A.: Bisimulation through probabilistic testing. Inf. Comput. **94**(1), 1–28 (1991)
35. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS (LNAI), vol. 6355, pp. 348–370. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_20
36. Lengál, O., Lin, A.W., Majumdar, R., Rümmer, P.: Fair termination for parameterized probabilistic concurrent systems. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10205, pp. 499–517. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_29
37. Lin, A.W., Rümmer, P.: Liveness of randomised parameterised systems under arbitrary schedulers. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 112–133. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_7
38. Löding, C., Madhusudan, P., Neider, D.: Abstract learning frameworks for synthesis. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 167–185. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_10
39. Milner, R.: Communication and Concurrency. Prentice Hall, Upper Saddle River (1989)
40. Neider, D., Jansen, N.: Regular model checking using solver technologies and automata learning. In: Brat, G., Rungta, N., Venet, A. (eds.) NFM 2013. LNCS, vol. 7871, pp. 16–31. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38088-4_2
41. Neider, D., Topcu, U.: An automaton learning approach to solving safety games over infinite graphs. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 204–221. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_12
42. Nilsson, M.: Regular model checking. Ph.D. thesis, Uppsala Universitet (2005)
43. Padon, O., Losa, G., Sagiv, M., Shoham, S.: Paxos made EPR: decidable reasoning about distributed protocols. In: PACMPL, 1(OOPSLA), pp. 108:1–108:31 (2017)
44. Padon, O., McMillan, K.L., Panda, A., Sagiv, M., Shoham, S.: Ivy: safety verification by interactive generalization. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, 13–17 June 2016, pp. 614–630 (2016)
45. PRISM case study: Dining Cryptographers. <http://www.prismmodelchecker.org/casestudies/diningcrypt.php>
46. Rubin, S.: Automatic structures. Ph.D. thesis, University of Auckland, New Zealand (2004)
47. Séniorgues, G.: The bisimulation problem for equational graphs of finite out-degree. SIAM J. Comput. **34**(5), 1025–1106 (2005)
48. Srba, J.: Roadmap of Infinite Results. Formal Models and Semantics, vol. 2. World Scientific Publishing Co., Singapore (2004)
49. To, A.W.: Model checking infinite-state systems: generic and specific approaches. Ph.D. thesis, LFCS, School of Informatics, University of Edinburgh (2010)
50. To, A.W., Libkin, L.: Recurrent reachability analysis in regular model checking. In: Cervesato, I., Veith, H., Voronkov, A. (eds.) LPAR 2008. LNCS (LNAI), vol. 5330, pp. 198–213. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89439-1_15
51. To, A.W., Libkin, L.: Algorithmic metatheorems for decidable LTL model checking over infinite systems. In: Ong, L. (ed.) FoSSaCS 2010. LNCS, vol. 6014, pp. 221–236. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12032-9_16

52. Valmari, A., Franceschinis, G.: Simple $O(m \log n)$ time Markov chain lumping. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 38–52. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12002-2_4
53. Vardhan, A.: Learning to verify systems. Ph.D. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign (2006)
54. Vardhan, A., Sen, K., Viswanathan, M., Agha, G.: Learning to verify safety properties. In: Davies, J., Schulte, W., Barnett, M. (eds.) ICFEM 2004. LNCS, vol. 3308, pp. 274–289. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30482-1_26

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Semi-quantitative Abstraction and Analysis of Chemical Reaction Networks

Milan Češka¹✉ and Jan Křetínský²

¹ Brno University of Technology, FIT,
IT4I Centre of Excellence, Brno, Czech Republic
ceskam@fit.vutbr.cz

² Technical University of Munich, Munich, Germany



Abstract. Analysis of large continuous-time stochastic systems is a computationally intensive task. In this work we focus on population models arising from chemical reaction networks (CRNs), which play a fundamental role in analysis and design of biochemical systems. Many relevant CRNs are particularly challenging for existing techniques due to complex dynamics including stochasticity, stiffness or multimodal population distributions. We propose a novel approach allowing not only to predict, but also to explain both the transient and steady-state behaviour. It focuses on qualitative description of the behaviour and aims at quantitative precision only in orders of magnitude. First we build a compact understandable model, which we then crudely analyse. As demonstrated on complex CRNs from literature, our approach reproduces the known results, but in contrast to the state-of-the-art methods, it runs with virtually no computational cost and thus offers unprecedented scalability.

1 Introduction

Chemical Reaction Networks (CRNs) are a versatile language widely used for *modelling and analysis* of biochemical systems [12] as well as for high-level *programming* of molecular devices [8, 40]. They provide a compact formalism equivalent to Petri nets [37], Vector Addition Systems (VAS) [29] and distributed population protocols [3]. Motivated by numerous potential applications ranging from system biology to synthetic biology, various techniques allowing simulation and formal analysis of CRNs have been proposed [2, 9, 21, 24, 39], and embodied in the design process of biochemical systems [20, 25, 32]. The time-evolution of CRNs is governed by the Chemical Master Equation (CME), which describes the probability of the molecular counts of each chemical species. Many important biochemical systems lead to complex dynamics that includes *state space explosion, stochasticity, stiffness, and multimodality* of the population distributions

This work has been supported by the Czech Science Foundation grant No. GA19-24397S, the IT4Innovations excellence in science project No. LQ1602, and the German Research Foundation (DFG) project KR 4890/2-1 “Statistical Unbounded Verification”.

© The Author(s) 2019

I. Dillig and S. Tasiran (Eds.): CAV 2019, LNCS 11561, pp. 475–496, 2019.

https://doi.org/10.1007/978-3-030-25540-4_28

[23, 44], and that fundamentally limits the class of systems the existing techniques can effectively handle. More importantly, biologist and engineers often seek for plausible explanations why the system under study has or has not the required behaviour. In many cases, a set of system simulations/trajectories or population distributions is not sufficient and the ability to provide an accurate explanation for the temporal or steady-state behaviour is another major challenge for the existing techniques.

In order to cope with the computational complexity of the analysis and in order to obtain explanations of the behaviour, we shift the focus from quantitatively precise results to a more qualitative analysis, closer to how a human would behold the system. Yet we insist on providing at least rough timing information on the behaviour as well as rough classification of probability of different behaviours at the extent of “very likely”, “few percent”, “barely possible”, so that we can conclude on issues such as time to extinction or bimodality of behaviour. This gives rise to our *semi-quantitative* approach. We stipulate that analyses in this framework reflect quantities in orders of magnitude, both for time duration and probabilities, but not more than that. This paradigm shift is reflected on two levels: (1) We abstract systems into semi-quantitative models. (2) We analyse systems in a semi-quantitative way. While each of the two can be combined with a traditional abstraction/analysis, when combined together they provide powerful means to understand systems’ behaviour with virtually no computational cost.

Semi-quantitative Models. The states of the models contain information on the current amount of objects of each species as an interval spanning often several orders of magnitude, unless instructed otherwise. For instance, if an amount of a certain species is to be closely monitored (as a part of the input specification/property of the system) then this abstraction can be finer. Similarly, whenever the analysis of a previous version of the abstraction points to the lack of precision in certain states, preventing us to conclude which of the possible behaviours is prevalent, the corresponding refinement can take place. Further, the rates of the transitions are also captured only with such imprecision. The crucial point allowing for existence of such models that are small, yet faithful, is our concept of *acceleration*. It captures certain *sequences* of transitions. It eliminates most of the non-determinism that paralyses other types of abstractions, which are too over-approximative, unable to conclude anything, but safety properties.

Semi-quantitative Analysis. Instead of performing exact transient or steady-state analysis, we can consider most probable transitions and then carefully lift this to most probable temporal behaviours. Technically, this is done by *alternating between transient and steady-state analysis* where only some rates and transitions are taken into account at different stages. In order to further facilitate the resulting insight of the human on the result of the analysis, we provide an algorithm to perform this analysis with virtually no computation effort and thus possibly manually. The trivial computations immediately pinpoint why certain

behaviours occur. Moreover, less likely behaviours can also be identified easily, to any desired degree of improbability (dozens of percent, promilles etc.).

To summarise, the first step yields tiny models, allowing for a synoptic observation of the model; due to their size these models can be either analysed easily using standard means, or can be subject to the second step. The second step provides an efficient approximative analysis, which is also very illustrative due to the limited use of quantities. It can be applied to any system; however, it is particularly interesting in connection with the models coming from the first step since (i) no extra effort (size, computation) is wasted on overly precise treatment that is ignored by the other step, and (ii) together they yield an understandable explanation of the behaviour. An entertaining feature of this paradigm is that the stiffer (with rates at hugely different time scales) the system is the easier it is to analyse.

To demonstrate the capabilities of our approach, we consider three challenging and biologically relevant case studies that have been used in literature to evaluate state-of-the-art methods for the CRN analysis. It has been shown that many approaches fail, either due to time-outs or incapability to capture differences in behaviours, and some tailored ones require considerable computational effort, e.g. an hour of computation. Our experiments clearly show that the proposed approach can deliver results that yield qualitatively same information, more understanding and can be computed in minutes by hand (or within a fraction of a second by computer).

Our contribution can be summarized as follows:

- We propose a novel *semi-quantitative* framework for analysis of CRN and similar population models, focusing on explainability of the results and low complexity, with quantitative precision limited to orders of magnitude.
- An algorithm for abstracting CRNs into semi-quantitative models based on interval abstraction of the species population and on transition acceleration.
- An algorithm for semi-quantitative analysis that replaces exact numerical computation by exploring the most probable transitions and alternating transient and steady-state analysis.
- We consider three challenging CRNs thoroughly studied in literature and demonstrate that the semi-quantitative abstraction and analysis gives us a unique tool that is able to accurately predict and explain both transient and steady-state behaviour of complex CRNs in a fraction of a second.

Related Work

To the best of our knowledge, there does not exist any abstraction of CRNs similar to the proposed approach. Indeed, there exist various abstraction and approximation schemes for CRNs that improve the performance and scalability of both the simulation-based and the numerical-based techniques. In the following paragraphs, we discuss the most relevant directions and the links to our approach.

Approximate Semantics for CRNs. For CRNs including large populations of species, fluid (mean-field) approximation techniques can be applied [5] and extended to approximate higher-order moments [15]: these deterministic approximations lead to a set of ordinary differential equations (ODEs). An alternative is to approximate the CME as a continuous-state stochastic process. The Linear Noise Approximation (LNA) is a Gaussian process which has been derived as an approximation of the CME [16, 44] and describes the time evolution of expectation and variance of the species in terms of ODEs. Recently, an aggregation scheme over ODEs that aims at understanding the dynamics of large CRNs has been proposed in [10]. In contrast to our approach, the deterministic approximations cannot adequately capture the stochasticity of CRNs caused by low population species.

To mitigate this drawback, various *hybrid models* have been proposed. The common idea of these models is as follows: the dynamics of low population species is described by the discrete stochastic process and the dynamics of large population species is approximated by a continuous process. The particular hybrid models differ in the approximation of the large population species. In [27], a pure deterministic semantics for large population species is used. The moment-based description for medium/high-copy number species was used in [24]. The LNA approximation and an adaptive partitioning of the species according to leap conditions (that is more general than partitioning based on population thresholds) was proposed in [9]. All hybrid models have to deal with interactions between low and large population species. In particular, the dynamics of the stochastic process describing the low-population species is conditioned by the continuous-state describing the concentration of the large-population species. The numerical analysis of such conditioned stochastic process is typically a computationally demanding task that limits the scalability.

In contrast, our approach does not explicitly partition the species, but rather abstracts the concrete species population using an interval abstraction and tries to effectively capture both the stochastic and the deterministic behaviour with the help of the accelerated transitions. As we already emphasised, the proposed abstraction and analysis avoids any numerical computation of precise quantities.

Reduction Techniques for Stochastic Models. A widely studied reduction method for Markov models is state aggregation based on lumping [6] or (bi-)simulation equivalence [4], with the latter notion in its exact [33] or approximate [13] form. Approximate notions of equivalence have led to new abstraction/refinement techniques for the numerical verification of Markov models over finite [14] as well as uncountably-infinite state spaces [1, 41, 42]. Several approximate aggregation schemes leveraging the structural properties of CRNs were proposed [17, 34, 45]. Abate et al. proposed an adaptive aggregation that gives formal guarantees on the approximation error, but typically provide lower state space reductions [2]. Our approach shares the idea of abstracting the state space by aggregating some states together. Similarly to [17, 34, 45], we partition the state space based on the species population, i.e. we also introduce the population levels. In contrast to the aforementioned aggregation schemes, we propose a

novel abstraction of the transition relation based on the acceleration. It allows us to avoid the numerical solution of the approximate CME and thus achieve a better reduction while providing an accurate predication of the system behaviour.

Alternative methods to deal with large/infinite state spaces are based on a state truncation trying to eliminate insignificant states, i.e., states reached only with a negligible probability. These methods, including finite state projections [36], sliding window abstractions [26], or fast adaptive uniformisation [35], are able to quantify the total probability mass that is lost due to the truncation, but typically cannot effectively handle systems involving a stiff behaviour and multimodality [9].

Simulation-Based Analysis. Transient analysis of CRNs can be performed using the Stochastic Simulation Algorithm (SSA) [21]. Note that the SSA produces a single realisation of the stochastic process, whereas the stochastic solution of CME gives the probability distribution of each species over time. Although simulation-based analysis is generally faster than direct solution of the stochastic process underlying the given CRN, obtaining good accuracy necessitates potentially large numbers of simulations and can be very time consuming.

Various partitioning schemes for species and reactions have been proposed for the purpose of speeding up the SSA in multi-scale systems [23, 38, 39]. For instance, Yao et al. introduced the slow-scale SSA [7], where they distinguish between fast and slow species. Fast species are then treated assuming they reach equilibrium much faster than the slow ones. Adaptive partitioning of the species has been considered in [19, 28]. In contrast to the simulation-based analysis, our approach (i) provides a compact explanation of the system behaviour in the form of tiny models allowing for a synoptic observation and (ii) can easily reveal less probable behaviours.

2 Chemical Reaction Networks

In this paper, we assume familiarity with standard verification of (continuous-time) probabilistic systems, e.g. [4]. For more detail, see [11, Appendix].

CRN Syntax. A *chemical reaction network (CRN)* $\mathcal{N} = (\Lambda, \mathcal{R})$ is a pair of finite sets, where Λ is a set of *species*, $|\Lambda|$ denotes its size, and \mathcal{R} is a set of reactions. Species in Λ interact according to the reactions in \mathcal{R} . A *reaction* $\tau \in \mathcal{R}$ is a triple $\tau = (r_\tau, p_\tau, k_\tau)$, where $r_\tau \in \mathbb{N}^{|\Lambda|}$ is the *reactant complex*, $p_\tau \in \mathbb{N}^{|\Lambda|}$ is the *product complex* and $k_\tau \in \mathbb{R}_{>0}$ is the coefficient associated with the rate of the reaction. r_τ and p_τ represent the stoichiometry of reactants and products. Given a reaction $\tau_1 = ([1, 1, 0], [0, 0, 2], k_1)$, we often refer to it as $\tau_1 : \lambda_1 + \lambda_2 \xrightarrow{k_1} 2\lambda_3$.

CRN Semantics. Under the usual assumption of mass action kinetics, the *stochastic semantics* of a CRN \mathcal{N} is generally given in terms of a discrete-state, continuous-time stochastic process $\mathbf{X}(\mathbf{t}) = (X_1(t), X_2(t), \dots, X_{|\Lambda|}(t), t \geq 0)$ [16]. The *state change* associated to the reaction τ is defined by $v_\tau = p_\tau - r_\tau$, i.e. the state \mathbf{X} is changed to $\mathbf{X}' = \mathbf{X} + v_\tau$, which we denote as $\mathbf{X} \xrightarrow{\tau} \mathbf{X}'$. For example,

for τ_1 as above, we have $v_{\tau_1} = [-1, -1, 2]$. For a reaction to happen in a state \mathbf{X} , all reactants have to be in sufficient numbers. The *reachable state space* of $\mathbf{X}(t)$, denoted as \mathbf{S} , is the set of all states reachable by a sequence of reactions from a given *initial state* \mathbf{X}_0 . The set of reactions changing the state \mathbf{X}_i to the state \mathbf{X}_j is denoted as $\text{reac}(\mathbf{X}_i, \mathbf{X}_j) = \{\tau \mid \mathbf{X}_i \xrightarrow{\tau} \mathbf{X}_j\}$.

The behaviour of the stochastic system $\mathbf{X}(t)$ can be described by the (possibly infinite) continuous-time Markov chain (CTMC) $\gamma(\mathcal{N}) = (\mathbf{S}, \mathbf{X}_0, \mathbf{R})$ where the transition matrix $\mathbf{R}(i, j)$ gives the probability of a transition from \mathbf{X}_i to \mathbf{X}_j . Formally,

$$\mathbf{R}(i, j) = \sum_{\tau \in \text{reac}(\mathbf{X}_i, \mathbf{X}_j)} k_\tau \cdot C_{\tau, i} \quad \text{where} \quad C_{\tau, i} = \prod_{\ell=1}^N \binom{\mathbf{X}_{i,\ell}}{r_\ell} \quad (\text{R})$$

corresponds to the population dependent term of the *propensity function* where $\mathbf{X}_{i,\ell}$ is ℓ th component of the state \mathbf{X}_i and r_ℓ is the stoichiometric coefficient of the ℓ -th reactant in the reaction τ . The CTMC $\gamma(\mathcal{N})$ is the accurate representation of CRN \mathcal{N} , but—even when finite—not scalable in practice because of the state space explosion problem [25, 31].

3 Semi-quantitative Abstraction

In this section, we describe our abstraction. We derive the desired CTMC conceptually in several steps, which we describe explicitly, although we implement the construction of the final system directly from the initial CRN.

3.1 Over-Approximation by Interval Abstraction and Acceleration

Given a CRN $\mathcal{N} = (\Lambda, \mathcal{R})$, we first consider an interval continuous-time Markov decision process (interval CTMDP¹), which is a finite abstraction of the infinite $\gamma(\mathcal{N})$. Intuitively, abstract states are given by intervals on sizes of populations with an additional specific that the abstraction captures enabledness of reactions. The transition structure follows the ideas of the standard may abstraction and of the three-valued abstraction of continuous-time systems [30]. A technical difference in the latter point is that we abstract rates into intervals instead of uniformising the chain and then only abstracting transition probabilities into intervals; this is necessary in later stages of the process. The main difference is that we also treat certain sequences of actions, which we call acceleration.

Abstract Domains. The first step is to define the abstract domain for the population sizes. For every species $\lambda \in \Lambda$, we define a finite partitioning A_λ of \mathbb{N} into intervals, reflecting the rough size of the population. Moreover, we want the abstraction to reflect whether a reaction is enabled. Hence we require that

¹ Interval CTMDP is a CTMDP with lower/upper bounds on rates. Since it serves only as an intermediate formalism to ease the presentation, we refrain from formalising it here.

$\{0\} \in A_\lambda$ for the case when the coefficients of this species as a reactant is always 0 or 1; in general, for every $i < \max_{\tau \in \mathcal{R}} r_\tau(\lambda)$ we require $\{i\} \in A_\lambda$.

The abstraction $\alpha_\lambda(n)$ of a number n of a species λ is then the $I \in A_\lambda$ for which $n \in I$. The state space of $\alpha(\mathcal{N})$ is the product $\prod_{\lambda \in \Lambda} A_\lambda$ of the abstract domains with the point-wise defined abstraction $\alpha(\mathbf{n})_\lambda = \alpha_\lambda(n_\lambda)$.

The abstract domain for the rates according to (R) is the set of all real intervals.

Transitions from an abstract state are defined as the may abstraction as follows. Since our abstraction reflect enabledness, the same set of action is enabled in all concrete states of a given abstract state. The targets of the action in the abstract setting are abstractions of all possible concrete successors, i.e. $\text{succ}(s, a) := \{\alpha(\mathbf{n}) \mid \mathbf{m} \in s, \mathbf{m} \xrightarrow{a} \mathbf{n}\}$, in other words, the transitions enabled in at least one of the respective concrete states. The abstract rate is the smallest interval including all the concrete rates of the respective concrete transitions. This can be easily computed by the corner-points abstraction (evaluating only the extremum values for each species) since the stoichiometry of the rates is monotone in the population sizes.

High-Level of Non-determinism. The (more or less) standard style of the abstraction above has several drawbacks—mostly related to the high degree of non-determinism for rates—which we will subsequently discuss.

Firstly, in connection with the abstract population sizes, transitions to different sizes only happen non-deterministically, leaving us unable to determine which behaviour is probable. For example, consider the simple system given by $\lambda \xrightarrow{d} \emptyset$ with $k_d = 10^{-4}$ so the degradation happens on average each 10^4 seconds. Assume population discretisation into $[0], [1..5], [6..20], [21..\infty)$ with abstraction depicted in Fig. 1. While the original system obviously moves from $[6..20]$ to $[1..5]$ very probably in less than $15 \cdot 10^4$ seconds, the abstraction cannot even say that it happens, not to speak of estimating the time.

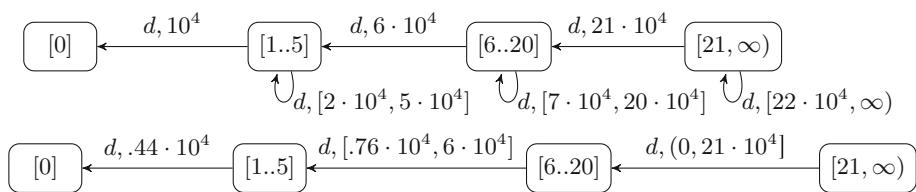


Fig. 1. Above: Interval CTMDP abstraction with intervals on rates and non-determinism. Below: Interval CTMC abstraction arising from acceleration.

Acceleration. To address this issue, we drop the non-deterministic self-loops and transitions to higher/lower populations in the abstract system.² Instead,

² One can also preserve the non-determinism for the special case when one of the transitions leads to a state where some action ceases to be enabled. While this adds more precision, the non-determinism in the abstraction makes it less convenient to handle.

we “*accelerate*” their effect: We consider sequences of these actions that in the concrete system have the effect of changing the population level. In our example above, we need to take the transition 1 to 13 times from [6..20] with various rates depending on the current concrete population, in order to get to [1..5]. This makes the precise timing more complicated to compute. Nevertheless, the expected time can be approximated easily: here it ranges from $\frac{1}{6} \cdot 10^4 = 0.17 \cdot 10^4$ (for population 6) to roughly $(\frac{1}{20} + \frac{1}{19} + \dots + \frac{1}{6}) \cdot 10^4 = 1.3 \cdot 10^4$ (for population 20). This results in an interval CTMC.³

Concurrency in Acceleration. The accelerated transitions can due to higher number of occurrences be considered continuous or deterministic, as opposed to discrete stochastic changes as distinguished in the hybrid approach. The usual differential equation approach would also take into account other reactions that are modelled deterministically and would combine their effect into one equation. In order to simplify the exposition and computation and—as we see later—without much loss of precision, we can consider only the fastest change (or non-deterministically more of them if their rates are similar).⁴

3.2 Operational Semantics: Concretisation to a Representative

The next disadvantage of classical abstraction philosophy, manifested in the interval CTMC above is that the precise-valued intervals on rates imply high computational effort during the analysis. Although the system is smaller, standard transient analysis is still quite expensive.

Concretisation. In order to deal with this issue, the interval can be approximated roughly by the expected time it would take for an average population in the considered range, in our example the “average” representative is 13. Then the first transition occurs with rate $13 \cdot 10^{-4} = 10^{-3}$ and needs to happen 7 times, yielding expected time $7/13 \cdot 10^4 = 0.5 \cdot 10^4$ (ignoring even the precise slow downs in the rates as the population decreases). Already this very rough computation yields relative precision with factor 3 for all the populations in this interval, thus yielding the correct order of magnitude with virtually no effort. We lift the concretisation naturally to states and denote the concretisation of abstract state s by $\gamma(s)$. The complete procedure is depicted in Algorithm 1.

The concretisation is one of the main points where we deliberately drop a lot of quantitative information, while still preserving some to conclude on big quantitative differences. Of course, the precision improves with more precise abstract domains and also with higher differences on the original rates.

³ The waiting times are not distributed according to the rates in the intervals. It is only the expected waiting time (reciprocal of the rate) that is preserved. Nevertheless, for ease of exposition, instead of labelling the transitions with expected waiting times we stick to the CTMC style with the reciprocals and formally treat it as if the label was a real rate.

⁴ Typically the classical concurrency diamond appears and the effect of the other accelerated reactions happen just after the first one.

Algorithm 1. Semi-quantitative abstraction CTMC $\alpha(\mathcal{N})$

1: $A \leftarrow \prod_{\lambda \in A} A_\lambda$	▷ States
2: for $a \in A$ do	▷ Transitions
3: $c \leftarrow \gamma(a)$	▷ Concrete representative
4: for each τ enabled in c do	
5: $r \leftarrow \text{rate of } \tau \text{ in } c$	▷ According to (R)
6: $a' \leftarrow \alpha(c + v_\tau)$	▷ Successor
7: set $a \xrightarrow{\tau} a'$ with rate r	
8: for self-loop $a \xrightarrow{\tau} a$ do	▷ Accelerate self-loops
9: $n_\tau \leftarrow \min\{n \mid \alpha(c + n \cdot v_\tau) \neq a\}$	▷ the number of τ to change the abstract state
10: $a' \leftarrow \alpha(c + n_\tau \cdot v_\tau)$	▷ Acceleration successor
11: instead of the self-loop with rate r , set $a \xrightarrow{\tau} a'$ with rate $n_\tau \cdot r$	

It remains to determine the representative for the unbounded interval. In order to avoid infinity, we require an additional input for the analysis, which are deemed upper bounds on possible population of each species. In cases when any upper bound is hard to assume, we can analyse the system with a random one and see if the last interval is reachable with significant probability. If yes, then we need to use this upper bound as a new point in the interval partitioning and try a higher upper bound next time. In general, such conditions can be checked in the abstraction and their violation implies a recommendation to refine the abstract domains accordingly.

Orders-of-Magnitude Abstraction. Such an approximation is thus sufficient to determine most of the time whether the acceleration (sequence of actions) happens sooner or later than e.g. another reaction with rate 10^{-6} or 10^{-2} . Note that this *decision* gets more precise not only as we refine the population levels, but also as the system gets stiffer (the concrete values of the rates differ more), which are normally harder to analyse. For the ease of presentation in our case studies, we shall depict only the magnitude of the rates, i.e. the decadic logarithm rounded to an integer.

Non-determinism and Refinement. If two rates are close to each other, say of the same magnitude (or difference 1), such a rough computation (and rough population discretisation) is not precise enough to determine which of the reactions happens with high probability sooner. Both may be happening roughly at the same pace, or with more information we could conclude one of them is considerably faster. This introduces an uncertainty, showing different behaviours are possible depending on the exact quantities. This indicates points where refinement might be needed if more precise results are required. For instance, with rates of magnitudes 2 and 3, the latter should be happening most of the time, the former only with a few percent chance. If we want to know whether it is rather tens of percent or tenths of percent, we should refine the abstraction.

4 Semi-quantitative Analysis

In this section, we present an approximative analysis technique that describes the most probable transient and steady-state behaviour of the system (also with rough timing) and on demand also the (one or more orders of magnitude) less probable behaviours. As such it is robust in the sense that it is well suited to work with imprecise rates and populations. It is computationally easy (can be done in hand in time required for a computer by other methods), while still yielding significant quantitative results (“in orders of magnitude”). It does not provide exact error guarantees since computing them would be almost as expensive as the classical analysis. It only features trivial limit-style bounds: if the population abstraction gets more and more refined, the probabilities converge to those of the original system; further, the higher the separation between the rate magnitudes, the more precise the approximation is since the other factors (and thus the incurred imprecisions) play less significant role.

Intuitively, the main idea—similar to some multi-rate simulation techniques for stiff systems—is to “simulate” “fast” reactions until the steady state and then examine which slower reactions take place. However, “fast” does not mean faster than some constant, but faster than other transitions in a given state. In other words, we are not distinguishing fast and slow reactions, but tailor this to each state separately. Further, “simulation” is not really a stochastic simulation, but a deterministic choice of the fastest available transition. If a transition is significantly faster than others then this yields what a simulation would yield. When there are transitions with similar rates, e.g. with at most one order of magnitude difference, then both are taken into account as described in the following definition.

Pruned System. Consider the underlying graph of the given CTMC. If we keep only the outgoing transitions with the maximum rate in each state, we call the result *pruned*. If there is always (at most) one transition then the graph consists of several paths leading to cycles. In general when more transitions are kept, it has bottom strongly connected components (bottom SCCs, BSCCs) and some transient parts.

We generalise this concept to *n-pruning* that preserves all transitions with a rate that is not more than n orders of magnitude smaller than the maximum rate in the state. Then the pruning above is 0-pruning, 1-pruning preserves also transitions happening up to 10 times slower, which can thus still happen with dozens of percent, 2-pruning is relevant for analysis where behaviour occurring with units of percent is also tracked etc.

Algorithm Idea. Here we explain the idea of Algorithm 2. The transient parts of the pruned system describe the most probable behaviour from each state until the point where visited states start to repeat a lot (steady state of the pruned system). In the original system, the usual behaviour is then to stay in this SCC C until one of the pruned (slower) reactions occurs, say from state s to state t . This may bring us to a different component of the pruned graph and the analysis process repeats. However, t may also bring us back into C , in which case we stay

in the steady-state, which is basically the same as without the transition from s to t . Further, t might be in the transient part leading to C , in which case these states are added to C and the steady state changes a bit, spreading the distribution slightly also to the previously transient states. Finally, t might be leading us into a component D where this run was previous to visiting C . In that case, the steady-state distribution spreads over all the components visited between D and C , putting a probability mass to each with a different order of magnitude depending on all the (magnitudes of) sojourn times in the transient and steady-state phases on the way.

Using the macros defined in the algorithm, the correctness of the computations can be shown as follows. For the time spent in the transient phase (line 16), we consider the slowest sojourn time on the way times the number of such transitions; this is accurate since the other times are by order(s) of magnitude shorter, hence negligible. The steady-state distribution on a BSCC of the

Algorithm 2. Semi-quantitative analysis

```

1:  $W \leftarrow \emptyset$                                      ▷ worklist of SCCs to process
2: add {initial state} to  $W$  and assign iteration 0 to it ▷ artificial SCC to start the process
3: while  $W \neq \emptyset$  do
4:    $C \leftarrow \text{pop } W$                                 ▷ Compute and output steady state or its approximation
5:   steady-state of  $C$  is approximately  $\minStayingRate/(m \cdot stayingRate(\cdot))$ 
6:   if  $C$  has no exits then continue                ▷ definitely bottom SCC, final steady state
      ▷ Compute and output exiting transitions and the time spent in  $C$ 
7:    $exitStates \leftarrow \arg \min_C(stayingRate(\cdot)/exitingRate(\cdot))$           ▷ Probable exit points
8:    $minStayingRate \leftarrow \text{minimum rate in } C$ ,  $m \leftarrow \#\text{occurrences there}$ 
9:    $timeToExit \leftarrow stayingRate(s) \cdot m / (|exitStates| \cdot minStayingRate \cdot exitingRate(s))$ 
      for (arbitrary)  $s \in exitStates$ 
10:  for all  $s \in exitStates$  do                         ▷ Transient analysis
11:     $t \leftarrow \text{target of the exiting transition}$ 
12:     $T \leftarrow \text{SCCs reachable in the pruned graph from } t$ 
13:    thereby newly reached transitions get assigned iteration of  $C + 1$ 
14:    for  $D \in T$  do                                ▷ Compute and output time to get from  $t$  to  $D$ 
15:       $minRate \leftarrow \text{minimum rate on the way from } t \text{ to } D$ ,  $m \leftarrow \#\text{occurrences there}$ 
16:       $transTime \leftarrow m / minRate$                   ▷ Determine the new SCC
17:      if  $D = C$  then                            ▷ back to the current SCC
18:        add to  $W$  the union of  $C$  and the new transient path  $\tau$  from  $t$  to  $C$ 
19:        in later steady-state computation, the states of  $\tau$  will have probability
           smaller by a factor of  $stayingRate(s)/exitingRate(s)$ 
20:      else if  $D$  was previously visited then ▷ alternating between different SCCs
21:        add to  $W$  the merge of all SCCs visited between  $D$  and  $C$  (inclusively)
22:        in later steady-state computation, reflect all  $timeToExit$  and  $transTime$ 
           between  $D$  and  $C$ 
23:      else                                         ▷ new SCC
24:        add  $D$  to  $W$ 
```

MACROS:

$stayingRate(s)$ is the rate of transitions from s in the pruned graph

$exitingRate(s)$ is the maximum rate of transitions from s not in the pruned graph

pruned graph can be approximated by the $\minStayingRate/(m \cdot stayingRate(\cdot))$ on line 5. Indeed, it corresponds to the steady-state distribution if the BSCC is a cycle and the \minStayingRate significantly larger than other rates in the BSCC since then the return time for the states is approximately m/\minStayingRate and the sojourn time $1/stayingRate(\cdot)$. The component is exited from s with the proportion given by its steady-state distribution times the probability to take the exit during that time. The former is approximated above; the latter can be approximated by the density in 0, i.e. by $exitingRate(s)$, since the staying rate is significantly faster. Hence the candidates for exiting are maximising $exitingRate(\cdot)/stayingRate(\cdot)$ as on line 7. There are $|exitStates|$ candidates for exit and the time to exit the component by a particular candidate s is the expected number of visits before exit, i.e. $stayingRate(s) \cdot exitingRate(s)$ times the return time $m \cdot \minStayingRate$, hence the expression on line 9.

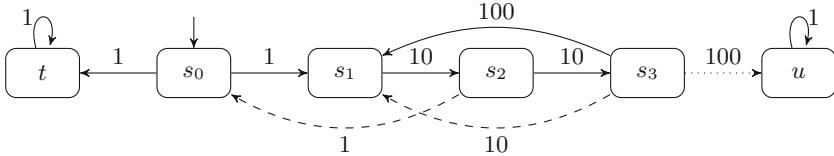


Fig. 2. Alternating transient and steady-state analysis.

For example, consider the system in Fig. 2. Iteration 1 reveals the part with solid lines with two (temporary) BSCCs $\{t\}$ and $\{s_1, s_2, s_3\}$. The former turns out definitely bottom. The latter has a steady state proportional to $(10^{-1}, 10^{-1}, 100^{-1})$. Its most probable exits are the dashed ones, identified in the subsequent iteration 2, probable proportionally to $(1/10, 10/100)$; the expected time to take them is $10 \cdot 2 / (2 \cdot 10 \cdot 1) = 1 = 100 \cdot 2 / (2 \cdot 10 \cdot 10)$. The latter leads back to the current SCC and does not change the set of BSCCs (hence in our examples below we often either skip or merge such iterations for the sake of readability). In contrast, the former leads to a previous SCC; thereafter $\{s_1, s_2, s_3\}$ is no more a bottom SCC and consequently the third exit to u is not even analysed. Nevertheless, it could still happen with minor probability, which can be seen if we consider 1-pruning instead.

5 Experimental Evaluation and Discussion

In order to demonstrate the applicability and accuracy of our approach, we selected the following three biologically relevant case studies. (1) stochastic model of gene expression [22, 24], (2) Goutsias's model [23] describing transcription regulation of a repressor protein in bacteriophage λ and (3) viral infection model [43].

Although the underlying CRNs are quite small (up to 5 species and 10 reaction), their analysis is very challenging: (i) the stochasticity has a strong impact

on the dynamics of these systems and thus purely deterministic approximations via ODEs are not accurate, (ii) the systems include species with low, medium, and high populations and thus the resulting state space of the stochastic process is prohibitively large to perform precise numerical analysis and existing reduction/approximation techniques are not sufficient (they are either too imprecise or do not provide sufficient reduction factors), and (iii) the system dynamics leads to bi-modal distributions and/or is affected by stiff reactions.

These models thus represent perfect candidates for evaluating advanced approximation methods including various hybrid approaches [9, 24, 27]. Although these approaches can handle the models, they typically require tens of minutes or hours of computation time. Similarly simulation-based methods are very time consuming especially in case of very stiff CRN, represented by the viral infection model. We demonstrate that our approach provides accurate predictions of the system behaviour and is feasible even when performed manually by a human.

Recall that the algorithm that builds the abstract model of the given CRN takes as input two vectors representing the population discretisation and population bounds. We generally assume that these inputs are provided by users who have a priori knowledge about the system (e.g. in which orders the species population occurs) and that the inputs also reflect the level of details the users are interested in. In the following case studies, we, however, set the inputs only based on the rate orders of the reactions affecting the particular species (unless mentioned otherwise).

5.1 Gene Expression Model

The CRN underlying the gene expression model is described in Table 1. As discussed in [24] and experimentally observed in [18], the system oscillates between two phases characterised by the D_{on} state and the D_{off} state, respectively. Biologists are interested in how the distribution of the D_{on} and D_{off} states is aligned with the distribution of RNA and proteins P, and how the correlation among the distributions depends on the DNA switching rates.

The state vector of the underlying CTMC is given as $[P, \text{RNA}, D_{\text{off}}, D_{\text{on}}]$. We use very relaxed bounds on the maximal populations, namely the bound 1000 for P and 100 for RNA. Note the DNA invariant $D_{\text{on}} + D_{\text{off}} = 1$. As in [24], the initial state is given as [10, 4, 1, 0].

We first consider the slow switching rates that lead to a more complicated dynamics including bimodal distributions. In order to demonstrate the refinement step and its effect on the accuracy of the model, we start with a very coarse abstraction. It distinguishes only the zero population and the non-zero populations and thus it is not able to adequately capture the relationship between the DNA state and RNA/P population. The pruned abstract model obtained using Algorithm 1 and 2 is depicted in Fig. 3 (left). The full one before pruning is shown in Fig. 6 [11, Appendix].

The proposed analysis of the model identifies the key trends in the system dynamic. The red transitions, representing iterations 1–3, capture the most probable paths in the system. The green component includes states with DNA on

Table 1. Gene expression. For slow DNA switching, $r_1 = r_2 = 0.05$. For fast DNA switching, $r_1 = r_2 = 1$. The rates are in h^{-1} .

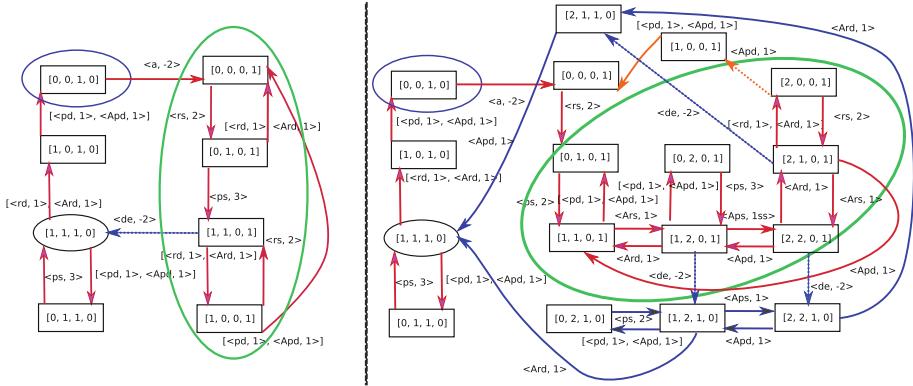
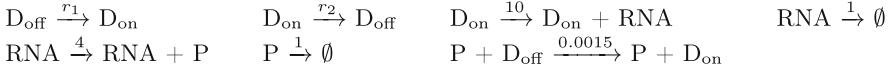


Fig. 3. Pruned abstraction for the gene expression model using the coarse population discretisation (left) and after the refinement (right). The state vector is $[P, \text{RNA}, D_{\text{off}}, D_{\text{on}}]$.

(i.e. $D_{\text{on}} = 1$) where the system oscillates. The component is reached via the blue state with D_{off} and no RNAs/P. The blue state is promptly reached from the initial state and then the system waits (roughly 100 h according our rate abstraction) for the next DNA activation. The oscillation is left via a deactivation in the iteration 4 (the blue dotted transition)⁵. The estimation of the exit time computed using Algorithm 2 is also 100 h. The deactivation is then followed by fast red transitions leading to the blue state, where the system waits for the next activation. Therefore, we obtain an oscillation between the blue state and the green component, representing the expected oscillation between the D_{on} and D_{off} states.

As expected, this abstraction does not clearly predict the bimodal distribution on the RNA/P populations as the trivial population levels do not bear any information beside reaction enabledness. In order to obtain a more accurate analysis of the system, we refine the population discretisation using a single level threshold for P and DNA, that is equal to 100 and 10, respectively (the rates in the CRN indicate that the population of P reaches higher values).

Figure 3 (right) depicts the pruned abstract model with the new discretisation (the full model is depicted in Fig. 7 [11, Appendix]. We again obtain the oscillation between the green component representing D_{on} states and the blue D_{off} state. The states in the green component more accurately predicts

⁵ In Fig. 3, the dotted transitions denote exit transitions representing the deactivations.

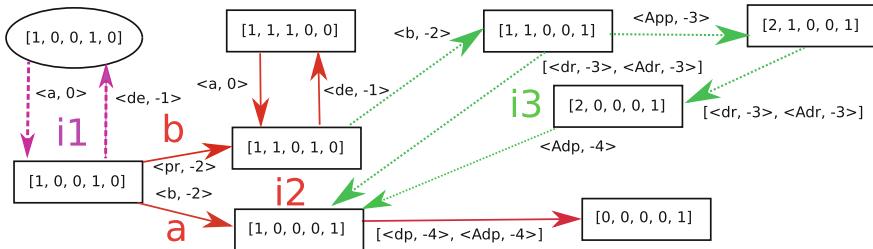
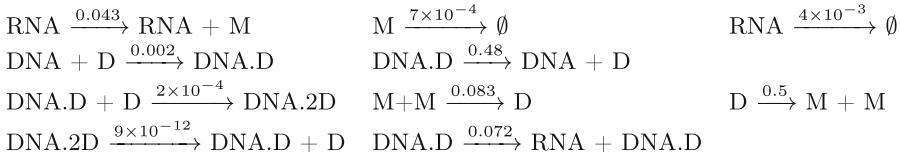
that in the DNA_{on} states the populations of RNA and P are high and drop to zero only for short time periods. The figure also shows orange transitions within the iteration 2 that extend the green component by two states. Note that the system promptly returns from these states back to the green component. After the deactivation in the iteration 4, the system takes (within the same iteration) the fast transitions (solid blue) leading to the blue component where system waits for another activation and where the mRNA/protein populations decrease. The expected time spent in states on blue solid transitions is small and thus we can reliably predict the bimodal distribution of the mRNA/P populations and its correlation with the DNA state. The refined abstraction also reveals that the switching time from the DNA_{on} mode to the DNA_{off} mode is lower. These predictions are in accordance with the results obtained in [24]. See Fig. 8 [11, Appendix] that is adopted from [24] and illustrates these results.

To further test the accuracy of our approach, we consider the fast switching between the DNA states. We follow the study in [24] and increase the rates by two orders of magnitude. We use the refined population discretisation and obtain a very similar abstraction as in Fig. 3 (right). We again obtain the oscillation between the green component (DNA_{on} states and nonzero RNA/protein populations) and the blue state (DNA_{off} and zero RNA/protein populations). The only difference is in fact the transition rates corresponding to the activation and deactivation causing that the switching rate between the components is much faster. As a consequence, the system spends a longer period in the blue transient states with D_{off} and nonzero RNA/protein populations. The time spent in these states decreases the correlation between the DNA state and the RNA/protein populations as well as the bimodality in the population distribution. This is again in the accordance with [24].

To conclude this case study, we observe a very aligned agreement between the results obtained using our approach and results in [24] obtained via advanced and time consuming numerical methods. We would like to emphasise that our abstraction and its solution is obtained within a fraction of a second while the numerical methods have to approximate solutions of equations describing high-order conditional moments of the population distributions. As [24] does not report the runtime of the analysis and the implementation of their methods is not publicly available, we cannot directly compare the time complexity.

5.2 Goutsias's Model

Goutsias's model illustrated in Table 2 is widely used for evaluation of various numerical and simulation based techniques. As showed e.g. in [23], the system has with a high probability the following transient behaviour. In the first phase, the system switches with a high rate between the non-active DNA (denoted DNA) and the active DNA (DNA.D). During this phase the population of RNA, monomers (M) and dimers (D) gradually increase (with only negligible oscillations). After around 15 min, the DNA is blocked (DNA.2D) and the population of RNA decreases while the population of M and D is relatively stable. After all RNA degrades (around another 15 min) the system switches to the third

Table 2. Goutsias' Model. The rates are in s^{-1} **Fig. 4.** Pruned abstraction for the Goutsias' model. The state vector is $[M + D, \text{RNA}, \text{DNA}, \text{DNA.D}, \text{DNA.2D}]$

phase where the population of M and D slowly decreases. Further, there is a non-negligible probability that the DNA is blocked at the beginning while the population of RNA is still small and the system promptly dies out.

Although the system is quite suitable for the hybrid approaches (there is no strong bimodality and only a limited stiffness), the analysis still takes 10 to 50 min depending on the required precision [27]. We demonstrate that our approach is able to accurately predict the main transient behaviour as well as the non-negligible probability that the system promptly dies out.

The state vector is given as $[M, D, \text{RNA}, \text{DNA}, \text{DNA.D}, \text{DNA.2D}]$ and the initial state is set to $[2, 6, 0, 1, 0, 0]$ as in [27]. We start our analysis with a coarse population discretisation with a single threshold 100 for M and D and a single threshold 10 for RNA. We relax the bounds, in particular, 1000 for M and D, and 100 for RNA. Note that these numbers were selected solely based on the rate orders of the relevant reactions. Note the DNA invariant $\text{DNA} + \text{DNA.D} + \text{DNA.2D} = 1$.

Figure 4 illustrates the pruned abstract model we obtained (the full model is depicted in Fig. 9 [11, Appendix]. For a better visualisation, we merged the state components corresponding to M and D into one component with $M + D$. As there is the fast reversible dimerisation, the actual distributions between the population of M and D does not affect the transient behaviour we are interested in.

The analysis of the model shows the following transient behaviour. The purple dotted loop in the iteration i1 represents (de-)activation of the DNA. The expected exit time of this loop is 100 s. According to our abstraction, there are two options (with the same probability) to exit the loop: (1) the path a rep-

resents the DNA blocking followed by the quick extinction and (2) the path b corresponds to the production of *RNA* and its followed by the red loop in the i_2 that again represents (de-)activation of the DNA. Note that according our abstraction, this loop contains states with the populations of M/D as well as RNA up to 100 and 10, respectively.

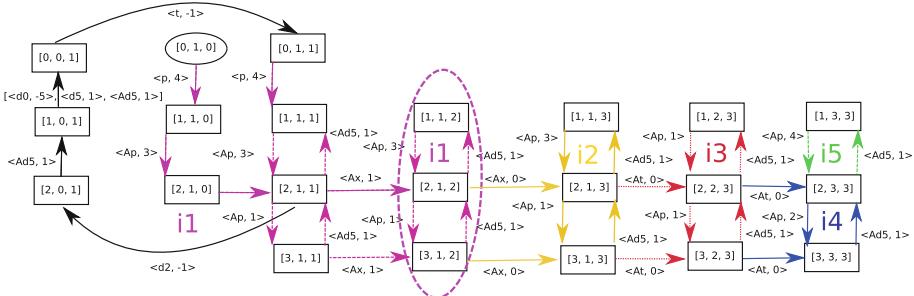
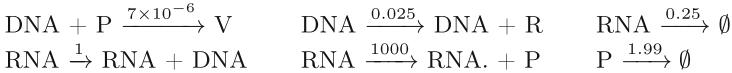
The expected exit time of this loop is again 100 s and there are two options how to leave the loop: (1) the path within the iteration i_3 (taken with roughly 90%) represents again the DNA blocking and it is followed by the extension of RNA and consequently by the extension of M/D in about 1000 s and (2) the path within the iteration 5 (shown in the full graph in Fig. 9 [11, Appendix]) taken with roughly 10% represents the series of protein productions and leads to the states with a high number of proteins (above 100 in our population discretisation). Afterwards, there is again a series of DNA (de-)activations followed by the DNA blocking and the extinction of RNA. As before, this leads to the extinction of M/D in about 1000 s.

Although this abstraction already shows the transient behaviour leading to the extinction in about 30 min, it introduces the following inaccuracy with respect to the known behaviour: (1) the probability of the fast extinction is higher and (2) we do not observe the clear bell-shape pattern on the RNA (i.e. the level 2 for the RNA is not reached in the abstraction). As in the previous case study, the problem is that the population discretisation is too coarse. It causes that the total rate of the DNA blocking (affected by the M/D population via the mass action kinetics) is too high in the states with the M/D population level 1. This can be directly seen in the interval CTMC representation where the rate spans many orders of magnitude, incurring too much imprecision. The refinement of the M/D population discretisation eliminates the first inaccuracy. To obtain the clear bell-shape patter on RNA, one has to refine also the RNA population discretisation.

5.3 Viral Infection

The viral infection model described in Table 3 represents the most challenging system we consider. It is highly stochastic, extremely stiff, with all species presenting high variance and some also very high molecular populations. Moreover, there is a bimodal distribution on the RNA population. As a consequence, the solution of the full CME, even using advanced reduction and aggregation techniques, is prohibitive due to state-space explosion and stochastic simulation are very time consuming. State-of-the-art hybrid approaches integrating the LNA and an adaptive population partitioning [9] can handle this system but also need a very long execution time. For example, a transient analysis up to time $t = 50$ requires around 20 min and up to $t = 200$ more than an hour.

To evaluate the accuracy of our approach on this challenging model, we also focus on the same transient analysis, namely, we are interested in the distribution of RNA at time $t = 200$. The analysis in [9] predicts a bimodal distribution where, the probability that RNA is zero in around 20% and the remaining probability has Gaussian distribution with mean around 17 and the probability that there

Table 3. Viral Infection. The rates are day⁻¹**Fig. 5.** Pruned abstraction for the viral infection model. The state vector is [P, RNA, DNA].

is more than 30 RNAs is close to zero. This is confirmed by simulation-based analysis in [23] showing also the gradual growth of the RNA population. The simulation-based analysis in [43], however, estimates a lower probability (around 3%) that RNA is 0 and higher mean of the remaining Gaussian distribution (around 23). Recall that obtaining accurate results using simulations is extremely time consuming due to very stiff reactions (a single simulation for $t = 200$ takes around 20 s).

In the final experiments, we analyse the distribution of RNA at time $t = 200$ using our approach. The state vector is given as [P, RNA, DNA] and we start with the concrete state [0, 1, 0]. To sufficiently reason about the RNA population and to handle the very high population of the proteins, we use the following population discretisation: thresholds {10, 1000} for P, {10, 30} for RNA, and {10, 100} for DNA. As before, we use very relaxed bounds 10000, 100, and 1000 for P, RNA, and D, respectively. Note that we ignore the population of the virus V as it does not affect the dynamics of the other species. This simplification makes the visualisation of our approach more readable and has no effect on the complexity of the analysis.

Figure 5 illustrates the obtained abstract model enabling the following transient analysis (the full model is depicted in Fig. 10 [11, Appendix]). In a few days the system reaches from the initial state the loop (depicted by the purple dashed ellipse) within the iteration $i1$. The loop includes states where RNA has level 1, DNA has level 2 and P oscillates between the levels 2 and 3. Before entering the loop, there is a non-negligible probability (orders of percent) that the RNA drops to 0 via the full black branch that returns to transient part of the loop in $i1$. In this branch the system can also die out (not shown in this figure, see the full model) with probability in the order of tenths of percent.

The average exit time of the loop in $i1$ is in the order of 10 days and the system goes to the yellow loop within the iteration $i2$, where the DNA level is increased to 3 (RNA level is unchanged and P again oscillates between the levels 2 and 3). The average exit time of the loop in $i2$ is again in the order of 10 days and systems goes to the dotted red loop within iteration $i3$. The transition represents the sequence of RNA synthesis that leads to RNA level 2. P oscillates as before. Finally, the system leaves the loop in $i3$ (this takes another dozen days) and reaches RNA level 3 in iterations $i4$ and $i5$ where the DNA level remains at the level 3 and P oscillates. The iteration $i4$ and $i5$ thus roughly correspond to the examined transient time $t = 200$.

The analysis clearly demonstrates that our approach leads to the behaviour that is well aligned with the previous experiments. We observed growth of the RNA population with a non-negligible probability of its extinction. The concrete quantities (i.e. the probability of the extinction and the mean RNA population) are closer to the analysis in [43]. The quantities are indeed affected by the population discretisation and can be further refined. We would like to emphasise that in contrast to the methods presented in [9, 23, 43] requiring hours of intensive numerical computation, our approach can be done even manually on the paper.

References

1. Abate, A., Katoen, J.P., Lygeros, J., Prandini, M.: Approximate model checking of stochastic hybrid systems. *Eur. J. Control* **16**, 624–641 (2010)
2. Abate, A., Brim, L., Češka, M., Kwiatkowska, M.: Adaptive aggregation of Markov chains: quantitative analysis of chemical reaction networks. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 195–213. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_12
3. Angluin, D., Aspnes, J., Eisenstat, D., Ruppert, E.: The computational power of population protocols. *Distrib. Comput.* **20**(4), 279–304 (2007)
4. Baier, C., Katoen, J.P.: Principles of Model Checking. The MIT Press, Cambridge (2008)
5. Bortolussi, L., Hillston, J.: Fluid model checking. In: Koutny, M., Ulidowski, I. (eds.) CONCUR 2012. LNCS, vol. 7454, pp. 333–347. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32940-1_24
6. Buchholz, P.: Exact performance equivalence: an equivalence relation for stochastic automata. *Theor. Comput. Sci.* **215**(1–2), 263–287 (1999)
7. Cao, Y., Gillespie, D.T., Petzold, L.R.: The slow-scale stochastic simulation algorithm. *J. Chem. Phys.* **122**(1), 014116 (2005)
8. Cardelli, L.: Two-domain DNA strand displacement. *Math. Struct. Comput. Sci.* **23**(02), 247–271 (2013)
9. Cardelli, L., Kwiatkowska, M., Laurenti, L.: A stochastic hybrid approximation for chemical kinetics based on the linear noise approximation. In: Bartocci, E., Lio, P., Paoletti, N. (eds.) CMSB 2016. LNCS, vol. 9859, pp. 147–167. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45177-0_10
10. Cardelli, L., Tribastone, M., Tschaikowski, M., Vandin, A.: Maximal aggregation of polynomial dynamical systems. *Proc. Natl. Acad. Sci.* **114**(38), 10029–10034 (2017)

11. Češka, M., Křetínský, J.: Semi-quantitative abstraction and analysis of chemical reaction networks. Technical report abs/1905.09914, [arXiv.org](https://arxiv.org/abs/1905.09914) (2019)
12. Chellaboina, V., Bhat, S.P., Haddad, W.M., Bernstein, D.S.: Modeling and analysis of mass-action kinetics. *IEEE Control Syst. Mag.* **29**(4), 60–78 (2009)
13. Desharnais, J., Laviolette, F., Tracol, M.: Approximate analysis of probabilistic processes: logic, simulation and games. In: Quantitative Evaluation of SysTems (QEST), pp. 264–273. IEEE (2008)
14. D’Innocenzo, A., Abate, A., Katoen, J.P.: Robust PCTL model checking. In: Hybrid Systems: Computation and Control (HSCC), pp. 275–285. ACM (2012)
15. Engblom, S.: Computing the moments of high dimensional solutions of the master equation. *Appl. Math. Comput.* **180**(2), 498–515 (2006)
16. Ethier, S.N., Kurtz, T.G.: Markov Processes: Characterization and Convergence, vol. 282. Wiley, New York (2009)
17. Ferm, L., Löstedt, P.: Adaptive solution of the master equation in low dimensions. *Appl. Numer. Math.* **59**(1), 187–204 (2009)
18. Gandhi, S.J., Zenklusen, D., Lionnet, T., Singer, R.H.: Transcription of functionally related constitutive genes is not coordinated. *Nat. Struct. Mol. Biol.* **18**(1), 27 (2011)
19. Ganguly, A., Altintan, D., Koepll, H.: Jump-diffusion approximation of stochastic reaction dynamics: error bounds and algorithms. *Multiscale Model. Simul.* **13**(4), 1390–1419 (2015)
20. Giacobbe, M., Guet, C.C., Gupta, A., Henzinger, T.A., Paixão, T., Petrov, T.: Model checking gene regulatory networks. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 469–483. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_47
21. Gillespie, D.T.: Exact stochastic simulation of coupled chemical reactions. *J. Phys. Chem.* **81**(25), 2340–2361 (1977)
22. Golding, I., Paulsson, J., Zawilski, S.M., Cox, E.C.: Real-time kinetics of gene activity in individual bacteria. *Cell* **123**(6), 1025–1036 (2005)
23. Goutsias, J.: Quasiequilibrium approximation of fast reaction kinetics in stochastic biochemical systems. *J. Chem. Phys.* **122**(18), 184102 (2005)
24. Hasenauer, J., Wolf, V., Kazeroonian, A., Theis, F.: Method of conditional moments (MCM) for the chemical master equation. *J. Math. Biol.* 1–49 (2013). <https://doi.org/10.1007/s00285-013-0711-5>
25. Heath, J., Kwiatkowska, M., Norman, G., Parker, D., Tymchyshyn, O.: Probabilistic model checking of complex biological pathways. *Theor. Comput. Sci.* **391**(3), 239–257 (2008)
26. Henzinger, T.A., Mateescu, M., Wolf, V.: Sliding window abstraction for infinite Markov chains. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 337–352. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_27
27. Henzinger, T.A., Mikeev, L., Mateescu, M., Wolf, V.: Hybrid numerical solution of the chemical master equation. In: Computational Methods in Systems Biology (CMSB), pp. 55–65. ACM (2010)
28. Hepp, B., Gupta, A., Khammash, M.: Adaptive hybrid simulations for multiscale stochastic reaction networks. *J. Chem. Phys.* **142**(3), 034118 (2015)
29. Karp, R.M., Miller, R.E.: Parallel program schemata. *J. Comput. Syst. Sci.* **3**(2), 147–195 (1969)
30. Katoen, J.-P., Klink, D., Leucker, M., Wolf, V.: Three-valued abstraction for continuous-time Markov chains. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 311–324. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73368-3_37

31. Kwiatkowska, M., Thachuk, C.: Probabilistic model checking for biology. *Softw. Syst. Saf.* **36**, 165 (2014)
32. Lakin, M.R., Parker, D., Cardelli, L., Kwiatkowska, M., Phillips, A.: Design and analysis of DNA strand displacement devices using probabilistic model checking. *J. R. Soc. Interface* **9**(72), 1470–1485 (2012)
33. Larsen, K.G., Skou, A.: Bisimulation through probabilistic testing. *Inf. Comput.* **94**(1), 1–28 (1991)
34. Madsen, C., Myers, C., Roehner, N., Winstead, C., Zhang, Z.: Utilizing stochastic model checking to analyze genetic circuits. In: Computational Intelligence in Bioinformatics and Computational Biology (CIBCB), pp. 379–386. IEEE (2012)
35. Mateescu, M., Wolf, V., Didier, F., Henzinger, T.A.: Fast adaptive uniformization of the chemical master equation. *IET Syst. Biol.* **4**(6), 441–452 (2010)
36. Munsky, B., Khammash, M.: The finite state projection algorithm for the solution of the chemical master equation. *J. Chem. Phys.* **124**, 044104 (2006)
37. Murata, T.: Petri nets: properties, analysis and applications. *Proc. IEEE* **77**(4), 541–580 (1989)
38. Rao, C.V., Arkin, A.P.: Stochastic chemical kinetics and the quasi-steady-state assumption: application to the Gillespie algorithm. *J. Chem. Phys.* **118**(11), 4999–5010 (2003)
39. Salis, H., Kaznessis, Y.: Accurate hybrid stochastic simulation of a system of coupled chemical or biochemical reactions. *J. Chem. Phys.* **122**(5), 054103 (2005)
40. Soloveichik, D., Seelig, G., Winfree, E.: DNA as a universal substrate for chemical kinetics. *Proc. Natl. Acad. Sci. U. S. A.* **107**(12), 5393–5398 (2010)
41. Soudjani, S.E.Z., Abate, A.: Adaptive and sequential gridding procedures for the abstraction and verification of stochastic processes. *SIAM J. Appl. Dyn. Syst.* **12**(2), 921–956 (2013)
42. Esmaeil Zadeh Soudjani, S., Abate, A.: Precise approximations of the probability distribution of a Markov process in time: an application to probabilistic invariance. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 547–561. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_45
43. Srivastava, R., You, L., Summers, J., Yin, J.: Stochastic vs. deterministic modeling of intracellular viral kinetics. *J. Theor. Biol.* **218**(3), 309–321 (2002)
44. Van Kampen, N.G.: Stochastic Processes in Physics and Chemistry, vol. 1. Elsevier, New York (1992)
45. Zhang, J., Watson, L.T., Cao, Y.: Adaptive aggregation method for the chemical master equation. *Int. J. Comput. Biol. Drug Des.* **2**(2), 134–148 (2009)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





PAC Statistical Model Checking for Markov Decision Processes and Stochastic Games

Pranav Ashok, Jan Křetínský,
and Maximilian Weininger^(✉)



Technical University of Munich, Munich, Germany
maxi.weininger@tum.de

Abstract. Statistical model checking (SMC) is a technique for analysis of probabilistic systems that may be (partially) unknown. We present an SMC algorithm for (unbounded) reachability yielding probably approximately correct (PAC) guarantees on the results. We consider both the setting (i) with no knowledge of the transition function (with the only quantity required a bound on the minimum transition probability) and (ii) with knowledge of the topology of the underlying graph. On the one hand, it is the first algorithm for stochastic games. On the other hand, it is the first practical algorithm even for Markov decision processes. Compared to previous approaches where PAC guarantees require running times longer than the age of universe even for systems with a handful of states, our algorithm often yields reasonably precise results within minutes, not requiring the knowledge of mixing time.

1 Introduction

Statistical model checking (SMC) [YS02a] is an analysis technique for probabilistic systems based on

1. simulating finitely many finitely long runs of the system,
2. statistical analysis of the obtained results,
3. yielding a confidence interval/probably approximately correct (PAC) result on the probability of satisfying a given property, i.e., there is a non-zero probability that the bounds are incorrect, but they are correct with probability that can be set arbitrarily close to 1.

One of the advantages is that it can avoid the state-space explosion problem, albeit at the cost of weaker guarantees. Even more importantly, this technique is applicable even when the model is not known (*black-box* setting) or only

This research was funded in part by TUM IGSSE Grant 10.06 (PARSEC), the Czech Science Foundation grant No. 18-11193S, and the German Research Foundation (DFG) project KR 4890/2-1 “Statistical Unbounded Verification”.

© The Author(s) 2019

I. Dillig and S. Tasiran (Eds.): CAV 2019, LNCS 11561, pp. 497–519, 2019.

https://doi.org/10.1007/978-3-030-25540-4_29

qualitatively known (*grey-box* setting), where the exact transition probabilities are unknown such as in many cyber-physical systems.

In the basic setting of Markov chains [Nor98] with (time- or step-)bounded properties, the technique is very efficient and has been applied to numerous domains, e.g. biological [JCL+09, PGL+13], hybrid [ZPC10, DDL+12, EGF12, Lar12] or cyber-physical [BBB+10, CZ11, DDL+13] systems and a substantial tool support is available [JLS12, BDL+12, BCLS13, BHH12]. In contrast, whenever either (i) infinite time-horizon properties, e.g. reachability, are considered or (ii) non-determinism is present in the system, providing any guarantees becomes significantly harder.

Firstly, for *infinite time-horizon properties* we need a stopping criterion such that the infinite-horizon property can be reliably evaluated based on a finite prefix of the run yielded by simulation. This can rely on the complete knowledge of the system (*white-box* setting) [YCZ10, LP08], the topology of the system (*grey box*) [YCZ10, HJB+10], or a lower bound p_{\min} on the minimum transition probability in the system (*black box*) [DHKP16, BCC+14].

Secondly, for Markov decision processes (MDP) [Put14] with (non-trivial) *non-determinism*, [HMZ+12] and [LP12] employ reinforcement learning [SB98] in the setting of bounded properties or discounted (and for the purposes of approximation thus also bounded) properties, respectively. The latter also yields PAC guarantees.

Finally, for MDP with unbounded properties, [BFHH11] deals with MDP with spurious non-determinism, where the way it is resolved does not affect the desired property. The general non-deterministic case is treated in [FT14, BCC+14], yielding PAC guarantees. However, the former requires the knowledge of mixing time, which is at least as hard to compute; the algorithm in the latter is purely theoretical since before a single value is updated in the learning process, one has to simulate longer than the age of universe even for a system as simple as a Markov chain with 12 states having at least 4 successors for some state.

Our contribution is an SMC algorithm with PAC guarantees for (i) MDP and unbounded properties, which runs for realistic benchmarks [HKP+19] and confidence intervals in orders of minutes, and (ii) is the first algorithm for stochastic games (SG). It relies on different techniques from literature.

1. The increased practical performance rests on two pillars:
 - extending early detection of bottom strongly connected components in Markov chains by [DHKP16] to end components for MDP and simple end components for SG;
 - improving the underlying PAC Q-learning technique of [SLW+06]:
 - (a) learning is now model-based with better information reuse instead of model-free, but in realistic settings with the same memory requirements,
 - (b) better guidance of learning due to interleaving with precise computation, which yields more precise value estimates.
 - (c) splitting confidence over all relevant transitions, allowing for variable width of confidence intervals on the learnt transition probabilities.

2. The transition from algorithms for MDP to SG is possible via extending the over-approximating value iteration from MDP [BCC+14] to SG by [KKKW18].

To summarize, we give an anytime PAC SMC algorithm for (unbounded) reachability. It is the first such algorithm for SG and the first practical one for MDP.

Related Work

Most of the previous efforts in SMC have focused on the analysis of properties with *bounded* horizon [YS02a, SVA04, YKNP06, JCL+09, JLS12, BDL+12].

SMC of *unbounded* properties was first considered in [HLMP04] and the first approach was proposed in [SVA05], but observed incorrect in [HJB+10]. Notably, in [YCZ10] two approaches are described. The first approach proposes to terminate sampled paths at every step with some probability p_{term} and re-weight the result accordingly. In order to guarantee the asymptotic convergence of this method, the second eigenvalue λ of the chain and its mixing time must be computed, which is as hard as the verification problem itself and requires the complete knowledge of the system (white box setting). The correctness of [LP08] relies on the knowledge of the second eigenvalue λ , too. The second approach of [YCZ10] requires the knowledge of the chain's topology (grey box), which is used to transform the chain so that all potentially infinite paths are eliminated. In [HJB+10], a similar transformation is performed, again requiring knowledge of the topology. In [DHKP16], only (a lower bound on) the minimum transition probability p_{min} is assumed and PAC guarantees are derived. While unbounded properties cannot be analyzed without any information on the system, knowledge of p_{min} is a relatively light assumption in many realistic scenarios [DHKP16]. For instance, bounds on the rates for reaction kinetics in chemical reaction systems are typically known; for models in the PRISM language [KNP11], the bounds can be easily inferred without constructing the respective state space. In this paper, we thus adopt this assumption.

In the case with general *non-determinism*, one approach is to give the non-determinism a probabilistic semantics, e.g., using a uniform distribution instead, as for timed automata in [DLL+11a, DLL+11b, Lar13]. Others [LP12, HMZ+12, BCC+14] aim to quantify over all strategies and produce an ϵ -optimal strategy. In [HMZ+12], candidates for optimal strategies are generated and gradually improved, but “at any given point we cannot quantify how close to optimal the candidate scheduler is” (cited from [HMZ+12]) and the algorithm “does not in general converge to the true optimum” (cited from [LST14]). Further, [LST14, DLST15, DHS18] randomly sample compact representation of strategies, resulting in useful lower bounds if ϵ -Schedulers are frequent. [HPS+19] gives a convergent model-free algorithm (with no bounds on the current error) and identifies that the previous [SKC+14] “has two faults, the second of which also affects approaches [...] [HAK18, HAK19]”.

Several approaches provide SMC for MDPs and unbounded properties with *PAC guarantees*. Firstly, similarly to [LP08, YCZ10], [FT14] requires (1) the

mixing time T of the MDP. The algorithm then yields PAC bounds in time polynomial in T (which in turn can of course be exponential in the size of the MDP). Moreover, the algorithm requires (2) the ability to restart simulations also in non-initial states, (3) it only returns the strategy once all states have been visited (sufficiently many times), and thus (4) requires the size of the state space $|S|$. Secondly, [BCC+14], based on delayed Q-learning (DQL) [SLW+06], lifts the assumptions (2) and (3) and instead of (1) mixing time requires only (a bound on) the minimum transition probability p_{\min} . Our approach additionally lifts the assumption (4) and allows for running times faster than those given by T , even without the knowledge of T .

Reinforcement learning (without PAC bounds) for stochastic games has been considered already in [LN81, Lit94, BT99]. [WT16] combines the special case of almost-sure satisfaction of a specification with optimizing quantitative objectives. We use techniques of [KKKW18], which however assumes access to the transition probabilities.

2 Preliminaries

2.1 Stochastic Games

A *probability distribution* on a finite set X is a mapping $\delta : X \rightarrow [0, 1]$, such that $\sum_{x \in X} \delta(x) = 1$. The set of all probability distributions on X is denoted by $\mathcal{D}(X)$. Now we define turn-based two-player stochastic games. As opposed to the notation of e.g. [Con92], we do not have special stochastic nodes, but rather a probabilistic transition function.

Definition 1 (SG). A stochastic game (SG) is a tuple $G = (S, S_{\square}, S_{\circlearrowleft}, s_0, A, Av, \mathbb{T})$, where S is a finite set of *states* partitioned¹ into the sets S_{\square} and S_{\circlearrowleft} of states of the player *Maximizer* and *Minimizer*², respectively $s_0 \in S$ is the *initial state*, A is a finite set of *actions*, $Av : S \rightarrow 2^A$ assigns to every state a set of *available actions*, and $\mathbb{T} : S \times A \rightarrow \mathcal{D}(S)$ is a *transition function* that given a state s and an action $a \in Av(s)$ yields a probability distribution over *successor* states. Note that for ease of notation we write $\mathbb{T}(s, a, t)$ instead of $\mathbb{T}(s, a)(t)$.

A Markov decision process (MDP) is a special case of SG where $S_{\circlearrowleft} = \emptyset$. A Markov chain (MC) can be seen as a special case of an MDP, where for all $s \in S : |Av(s)| = 1$. We assume that SG are non-blocking, so for all states s we have $Av(s) \neq \emptyset$.

For a state s and an available action $a \in Av(s)$, we denote the set of successors by $\text{Post}(s, a) := \{t \mid \mathbb{T}(s, a, t) > 0\}$. We say a state-action pair (s, a) is an *exit* of a set of states T , written (s, a) exits T , if $\exists t \in \text{Post}(s, a) : t \notin T$, i.e., if with some probability a successor outside of T could be chosen.

We consider algorithms that have a limited information about the SG.

¹ I.e., $S_{\square} \subseteq S$, $S_{\circlearrowleft} \subseteq S$, $S_{\square} \cup S_{\circlearrowleft} = S$, and $S_{\square} \cap S_{\circlearrowleft} = \emptyset$.

² The names are chosen, because Maximizer maximizes the probability of reaching a given target state, and Minimizer minimizes it.

Definition 2 (Black box and grey box). An algorithm inputs an SG as black box if it cannot access the whole tuple, but

- it knows the initial state,
- for a given state, an oracle returns its player and available action,
- given a state s and action a , it can sample a successor t according to $\mathbb{T}(s, a)$,³
- it knows $p_{\min} \leq \min_{\substack{s \in S, a \in Av(s) \\ t \in Post(s, a)}} \mathbb{T}(s, a, t)$, an under-approximation of the minimum transition probability.

When input as grey box it additionally knows the number $|Post(s, a)|$ of successors for each state s and action a .⁴

The semantics of SG is given in the usual way by means of strategies and the induced Markov chain [BK08] and its respective probability space, as follows. An *infinite path* ρ is an infinite sequence $\rho = s_0 a_0 s_1 a_1 \dots \in (S \times A)^\omega$, such that for every $i \in \mathbb{N}$, $a_i \in Av(s_i)$ and $s_{i+1} \in Post(s_i, a_i)$.

A *strategy* of Maximizer or Minimizer is a function $\sigma : S_\square \rightarrow \mathcal{D}(A)$ or $S_\circlearrowleft \rightarrow \mathcal{D}(A)$, respectively, such that $\sigma(s) \in \mathcal{D}(Av(s))$ for all s . Note that we restrict to memoryless/positional strategies, as they suffice for reachability in SGs [CH12].

A pair (σ, τ) of strategies of Maximizer and Minimizer induces a Markov chain $G^{\sigma, \tau}$ with states S , s_0 being initial, and the transition function $\mathbb{T}(s)(t) = \sum_{a \in Av(s)} \sigma(s)(a) \cdot \mathbb{T}(s, a, t)$ for states of Maximizer and analogously for states of Minimizer, with σ replaced by τ . The Markov chain induces a unique probability distribution $\mathbb{P}^{\sigma, \tau}$ over measurable sets of infinite paths [BK08, Ch. 10].

2.2 Reachability Objective

For a goal set $Goal \subseteq S$, we write $\Diamond Goal := \{s_0 a_0 s_1 a_1 \dots \mid \exists i \in \mathbb{N} : s_i \in Goal\}$ to denote the (measurable) set of all infinite paths which eventually reach $Goal$. For each $s \in S$, we define the *value* in s as

$$V(s) := \sup_{\sigma} \inf_{\tau} \mathbb{P}_s^{\sigma, \tau}(\Diamond Goal) = \inf_{\tau} \sup_{\sigma} \mathbb{P}_s^{\sigma, \tau}(\Diamond Goal),$$

where the equality follows from [Mar75]. We are interested in $V(s_0)$, its ε -approximation and the corresponding (ε -)optimal strategies for both players.

³ Up to this point, this definition conforms to black box systems in the sense of [SVA04] with sampling from the initial state, being slightly stricter than [YS02a] or [RP09], where simulations can be run from any desired state. Further, we assume that we can choose actions for the adversarial player or that she plays fairly. Otherwise the adversary could avoid playing her best strategy during the SMC, not giving SMC enough information about her possible behaviours.

⁴ This requirement is slightly weaker than the knowledge of the whole topology, i.e. $Post(s, a)$ for each s and a .

Let Zero be the set of states, from which there is no finite path to any state in Goal . The value function V satisfies the following system of equations, which is referred to as the *Bellman equations*:

$$V(s) = \begin{cases} \max_{a \in \text{Av}(s)} V(s, a) & \text{if } s \in S_{\square} \\ \min_{a \in \text{Av}(s)} V(s, a) & \text{if } s \in S_{\bigcirc} \\ 1 & \text{if } s \in \text{Goal} \\ 0 & \text{if } s \in \text{Zero} \end{cases}$$

with the abbreviation $V(s, a) := \sum_{s' \in S} T(s, a, s') \cdot V(s')$. Moreover, V is the *least* solution to the Bellman equations, see e.g. [CH08].

2.3 Bounded and Asynchronous Value Iteration

The well known technique of value iteration, e.g. [Put14, RF91], works by starting from an under-approximation of value function and then applying the Bellman equations. This converges towards the least fixpoint of the Bellman equations, i.e. the *value function*. Since it is difficult to give a convergence criterion, the approach of bounded value iteration (BVI, also called interval iteration) was developed for MDP [BCC+14, HM17] and SG [KKKW18]. Beside the under-approximation, it also updates an over-approximation according to the Bellman equations. The most conservative over-approximation is to use an upper bound of 1 for every state. For the under-approximation, we can set the lower bound of target states to 1; all other states have a lower bound of 0. We use the function `INITIALIZE_BOUNDS` in our algorithms to denote that the lower and upper bounds are set as just described; see [AKW19, Algorithm 8] for the pseudocode. Additionally, BVI ensures that the over-approximation converges to the least fixpoint by taking special care of *end components*, which are the reason for not converging to the true value from above.

Definition 3 (End component (EC)). A non-empty set $T \subseteq S$ of states is an end component (EC) if there is a non-empty set $B \subseteq \bigcup_{s \in T} \text{Av}(s)$ of actions such that (i) for each $s \in T, a \in B \cap \text{Av}(s)$ we do not have (s, a) exits T and (ii) for each $s, s' \in T$ there is a finite path $w = sa_0 \dots a_n s' \in (T \times B)^* \times T$, i.e. the path stays inside T and only uses actions in B .

Intuitively, ECs correspond to bottom strongly connected components of the Markov chains induced by possible strategies, so for some pair of strategies all possible paths starting in the EC remain there. An end component T is a *maximal end component (MEC)* if there is no other end component T' such that $T \subseteq T'$. Given an SG G , the set of its MECs is denoted by $\text{MEC}(G)$.

Note that, to stay in an EC in an SG, the two players would have to cooperate, since it depends on the pair of strategies. To take into account the adversarial behaviour of the players, it is also relevant to look at a subclass of ECs, the so called *simple end components*, introduced in [KKKW18].

Definition 4 (Simple end component (SEC) [KKKW18]). An EC T is called simple, if for all $s \in T$ it holds that $V(s) = \text{bestExit}(T, V)$, where

$$\text{bestExit}(T, f) := \begin{cases} 1 & \text{if } T \cap \text{Goal} \neq \emptyset \\ \max_{\substack{s \in T \cap S_\square \\ (s,a) \text{ exits } T}} f(s, a) & \text{else} \end{cases}$$

is called the best exit (of Maximizer) from T according to the function $f : S \rightarrow \mathbb{R}$. To handle the case that there is no exit of Maximizer in T we set $\max_\emptyset = 0$.

Intuitively, SECs are ECs where Minimizer does not want to use any of her exits, as all of them have a greater value than the best exit of Maximizer. Assigning any value between those of the best exits of Maximizer and Minimizer to all states in the EC is a solution to the Bellman equations, because both players prefer remaining and getting that value to using their exits [KKKW18, Lemma 1]. However, this is suboptimal for Maximizer, as the goal is not reached if the game remains in the EC forever. Hence we “deflate” the upper bounds of SECs, i.e. reduce them to depend on the best exit of Maximizer. T is called maximal simple end component (MSEC), if there is no SEC T' such that $T \subsetneq T'$. Note that in MDPs, treating all MSECs amounts to treating all MECs.

Algorithm 1. Bounded value iteration algorithm for SG (and MDP)

```

1: procedure BVI(SG G, target set Goal, precision  $\epsilon > 0$ )
2:   INITIALIZE_BOUNDS
3:   repeat
4:      $X \leftarrow \text{SIMULATE until LOOPING or state in Goal is hit}$ 
5:     UPDATE( $X$ )                                 $\triangleright$  Bellman updates or their modification
6:     for  $T \in \text{FIND\_MSECs}(X)$  do
7:       DEFLATE( $T$ )                             $\triangleright$  Decrease the upper bound of MSECs
8:     until  $U(s_0) - L(s_0) < \epsilon$ 
```

Algorithm 1 rephrases that of [KKKW18] and describes the general structure of all bounded value iteration algorithms that are relevant for this paper. We discuss it here since all our improvements refer to functions (in capitalized font) in it. In the next section, we design new functions, pinpointing the difference to the other papers. The pseudocode of the functions adapted from the other papers can be found, for the reader’s convenience, in [AKW19, Appendix A]. Note that to improve readability, we omit the parameters G , Goal , L and U of the functions in the algorithm.

Bounded Value Iteration: For the standard bounded value iteration algorithm, Line 4 does not run a simulation, but just assigns the whole state space S to X ⁵. Then it updates all values according to the Bellman equations.

⁵ Since we mainly talk about simulation based algorithms, we included this line to make their structure clearer.

After that it finds all the problematic components, the MSECs, and “deflates” them as described in [KKKW18], i.e. it reduces their values to ensure the convergence to the least fixpoint. This suffices for the bounds to converge and the algorithm to terminate [KKKW18, Theorem 2].

Asynchronous Bounded Value Iteration: To tackle the state space explosion problem, *asynchronous* simulation/learning-based algorithms have been developed [MLG05, BCC+14, KKKW18]. The idea is not to update and deflate all states at once, since there might be too many, or since we only have limited information. Instead of considering the whole state space, a path through the SG is sampled by picking in every state one of the actions that look optimal according to the current over-/under-approximation and then sampling a successor of that action. This is repeated until either a target is found, or until the simulation is looping in an EC; the latter case occurs if the heuristic that picks the actions generates a pair of strategies under which both players only pick staying actions in an EC. After the simulation, only the bounds of the states on the path are updated and deflated. Since we pick actions which look optimal in the simulation, we almost surely find an ϵ -optimal strategy and the algorithm terminates [BCC+14, Theorem 3].

3 Algorithm

3.1 Model-Based

Given only limited information, updating cannot be done using \mathbb{T} , since the true probabilities are not known. The approach of [BCC+14] is to sample for a high number of steps and accumulate the observed lower and upper bounds on the true value function for each state-action pair. When the number of samples is large enough, the average of the accumulator is used as the new estimate for the state-action pair, and thus the approximations can be improved and the results back-propagated, while giving statistical guarantees that each update was correct. However, this approach has several drawbacks, the biggest of which is that the number of steps before an update can occur is infeasibly large, often larger than the age of the universe, see Table 1 in Sect. 4.

Our improvements to make the algorithm practically usable are linked to constructing a partial model of the given system. That way, we have more information available on which we can base our estimates, and we can be less conservative when giving bounds on the possible errors. The shift from model-free to model-based learning asymptotically increases the memory requirements from $\mathcal{O}(|S| \cdot |A|)$ (as in [SLW+06, BCC+14]) to $\mathcal{O}(|S|^2 \cdot |A|)$. However, for systems where each action has a small constant bound on the number of successors, which is typical for many practical systems, e.g. classical PRISM benchmarks, it is still $\mathcal{O}(|S| \cdot |A|)$ with a negligible constant difference.

We thus track the number of times some successor t has been observed when playing action a from state s in a variable $\#(s, a, t)$. This implicitly induces the number of times each state-action pair (s, a) has been played $\#(s, a) =$

$\sum_{t \in S} \#(s, a, t)$. Given these numbers we can then calculate probability estimates for every transition as described in the next subsection. They also induce the set of all states visited so far, allowing us to construct a partial model of the game. See [AKW19, Appendix A.2] for the pseudo-code of how to count the occurrences during the simulations.

3.2 Safe Updates with Confidence Intervals Using Distributed Error Probability

We use the counters to compute a lower estimate of the transition probability for some error tolerance δ_T as follows: We view sampling t from state-action pair (s, a) as a Bernoulli sequence, with success probability $T(s, a, t)$, the number of trials $\#(s, a)$ and the number of successes $\#(s, a, t)$. The tightest lower estimate we can give using the Hoeffding bound (see [AKW19, Appendix D.1]) is

$$\widehat{T}(s, a, t) := \max(0, \frac{\#(s, a, t)}{\#(s, a)} - c), \quad (1)$$

where the confidence width $c := \sqrt{\frac{\ln(\delta_T)}{-2\#(s, a)}}$. Since c could be greater than 1, we limit the lower estimate to be at least 0. Now we can give modified update equations:

$$\begin{aligned} \widehat{L}(s, a) &:= \sum_{t: \#(s, a, t) > 0} \widehat{T}(s, a, t) \cdot L(t) \\ \widehat{U}(s, a) &:= \left(\sum_{t: \#(s, a, t) > 0} \widehat{T}(s, a, t) \cdot U(t) \right) + \left(1 - \sum_{t: \#(s, a, t) > 0} \widehat{T}(s, a, t) \right) \end{aligned}$$

The idea is the same for both upper and lower bound: In contrast to the usual Bellman equation (see Sect. 2.2) we use \widehat{T} instead of T . But since the sum of all the lower estimates does not add up to one, there is some remaining probability for which we need to under-/over-approximate the value it can achieve. We use

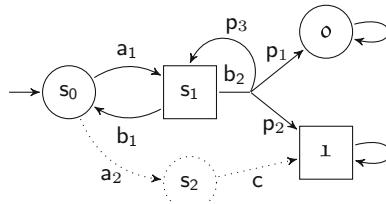


Fig. 1. A running example of an SG. The dashed part is only relevant for the later examples. For actions with only one successor, we do not depict the transition probability 1 (e.g. $T(s_0, a_1, s_1)$). For state-action pair (s_1, b_2) , the transition probabilities are parameterized and instantiated in the examples where they are used.

the safe approximations 0 and 1 for the lower and upper bound respectively; this is why in \mathcal{L} there is no second term and in $\widehat{\mathcal{U}}$ the whole remaining probability is added. Algorithm 2 shows the modified update that uses the lower estimates; the proof of its correctness is in [AKW19, Appendix D.2].

Lemma 1 (UPDATE is correct). *Given correct under- and over-approximations \mathcal{L}, \mathcal{U} of the value function V , and correct lower probability estimates $\widehat{\mathbb{T}}$, the under- and over-approximations after an application of UPDATE are also correct.*

Algorithm 2. New update procedure using the probability estimates

```

1: procedure UPDATE(State set  $X$ )
2:   for  $f \in \{\mathcal{L}, \mathcal{U}\}$  do                                 $\triangleright$  For both functions
3:     for  $s \in X \setminus \text{Goal}$  do                 $\triangleright$  For all non-target states in the given set
4:        $f(s) = \begin{cases} \max_{a \in \text{Av}(s)} \widehat{f}(s, a) & \text{if } s \in S_\square \\ \min_{a \in \text{Av}(s)} \widehat{f}(s, a) & \text{if } s \in S_\circ \end{cases}$ 

```

Example 1. We illustrate how the calculation works and its huge advantage over the approach from [BCC+14] on the SG from Fig. 1. For this example, ignore the dashed part and let $p_1 = p_2 = 0.5$, i.e. we have no self loop, and an even chance to go to the target $\mathbf{1}$ or a sink \mathbf{o} . Observe that hence $V(s_0) = V(s_1) = 0.5$.

Given an error tolerance of $\delta = 0.1$, the algorithm of [BCC+14] would have to sample for more than 10^9 steps before it could attempt a single update. In contrast, assume we have seen 5 samples of action b_2 , where 1 of them went to $\mathbf{1}$ and 4 of them to \mathbf{o} . Note that, in a sense, we were unlucky here, as the observed averages are very different from the actual distribution. The confidence width for $\delta_{\mathbb{T}} = 0.1$ and 5 samples is $\sqrt{\ln(0.1)} / -2 \cdot 5 \approx 0.48$. So given that data, we get $\widehat{\mathbb{T}}(s_1, b_2, \mathbf{1}) = \max(0, 0.2 - 0.48) = 0$ and $\widehat{\mathbb{T}}(s_1, b_2, \mathbf{o}) = \max(0, 0.8 - 0.48) = 0.32$. Note that both probabilities are in fact lower estimates for their true counterpart.

Assume we already found out that \mathbf{o} is a sink with value 0; how we gain this knowledge is explained in the following subsections. Then, after getting only these 5 samples, UPDATE already decreases the upper bound of (s_1, b_2) to 0.68, as we know that at least 0.32 of $\mathbb{T}(s_1, b_2)$ goes to the sink.

Given 500 samples of action b_2 , the confidence width of the probability estimates already has decreased below 0.05. Then, since we have this confidence width for both the upper and the lower bound, we can decrease the total precision for (s_1, b_2) to 0.1, i.e. return an interval in the order of $[0.45; 0.55]$. \triangleleft

Summing up: with the model-based approach we can already start updating after very few steps and get a reasonable level of confidence with a realistic number of samples. In contrast, the state-of-the-art approach of [BCC+14] needs a very large number of samples even for this toy example.

Since for UPDATE we need an error tolerance for every transition, we need to distribute the given total error tolerance δ over all transitions in the current

partial model. For all states in the explored partial model \hat{S} we know the number of available actions and can over-approximate the number of successors as $\frac{1}{p_{\min}}$. Thus the error tolerance for each transition can be set to $\delta_T := \frac{\delta \cdot p_{\min}}{|\{a|s \in \hat{S} \wedge a \in \text{Av}(s)\}|}$. This is illustrated in Example 4 in [AKW19, Appendix B].

Note that the fact that the error tolerance δ_T for every transition is the same does *not* imply that the confidence width for every transition is the same, as the latter becomes smaller with increasing number of samples $\#(s, a)$.

3.3 Improved EC Detection

As mentioned in the description of Algorithm 1, we must detect when the simulation is stuck in a bottom EC and looping forever. However, we may also stop simulations that are looping in some EC but still have a possibility to leave it; for a discussion of different heuristics from [BCC+14, KKW18], see [AKW19, Appendix A.3].

We choose to define LOOPING as follows: Given a candidate for a bottom EC, we continue sampling until we are δ_T -sure (i.e. the error probability is smaller than δ_T) that we cannot leave it. Then we can safely deflate the EC, i.e. decrease all upper bounds to zero.

To detect that something is a δ_T -sure EC, we do not sample for the astronomical number of steps as in [BCC+14], but rather extend the approach to detect bottom strongly connected components from [DHKP16]. If in the EC-candidate T there was some state-action pair (s, a) that actually has a probability to exit the T , that probability is at least p_{\min} . So after sampling (s, a) for n times, the probability to overlook such a leaving transition is $(1 - p_{\min})^n$ and it should be smaller than δ_T . Solving the inequation for the required number of samples n yields $n \geq \frac{\ln(\delta_T)}{\ln(1 - p_{\min})}$.

Algorithm 3 checks that we have seen all staying state-action pairs n times, and hence that we are δ_T -sure that T is an EC. Note that we restrict to staying state-action pairs, since the requirement for an EC is only that there exist staying actions, not that all actions stay. We further speed up the EC-detection, because we do not wait for n samples in every simulation, but we use the aggregated counters that are kept over all simulations.

Algorithm 3. Check whether we are δ_T -sure that T is an EC

```

1: procedure  $\delta_T$ -sure EC (State set  $T$ )
2:    $requiredSamples = \frac{\ln(\delta_T)}{\ln(1 - p_{\min})}$ 
3:    $B \leftarrow \{(s, a) \mid s \in T \wedge \neg(s, a) \text{ exits } T\}$             $\triangleright$  Set of staying state-action pairs
4:   return  $\bigwedge_{(s,a) \in B} \#(s, a) > requiredSamples$ 
```

We stop a simulation, if LOOPING returns true, i.e. under the following three conditions: (i) We have seen the current state before in this simulation ($s \in X$),

i.e. there is a cycle. (ii) This cycle is explainable by an EC T in our current partial model. (iii) We are $\delta_{\mathbb{T}}$ -sure that T is an EC.

Algorithm 4. Check if we are probably looping and should stop the simulation

```

1: procedure LOOPING(State set  $X$ , state  $s$ )
2:   if  $s \notin X$  then
3:     return false                                 $\triangleright$  Easy improvement to avoid overhead
4:   return  $\exists T \subseteq X. T \text{ is EC in partial model} \wedge s \in T \wedge \delta_{\mathbb{T}}\text{-sure } \text{EC}(T)$ 
```

Example 2. For this example, we again use the SG from Fig. 1 without the dashed part, but this time with $p_1 = p_2 = p_3 = \frac{1}{3}$. Assume the path we simulated is $(s_0, a_1, s_1, b_2, s_1)$, i.e. we sampled the self-loop of action b_2 . Then $\{s_1\}$ is a candidate for an EC, because given our current observation it seems possible that we will continue looping there forever. However, we do not stop the simulation here, because we are not yet $\delta_{\mathbb{T}}$ -sure about this. Given $\delta_{\mathbb{T}} = 0.1$, the required samples for that are 6, since $\frac{\ln(0.1)}{\ln(1 - \frac{1}{3})} = 5.6$. With high probability (greater than $(1 - \delta_{\mathbb{T}}) = 0.9$), within these 6 steps we will sample one of the other successors of (s_1, b_2) and thus realise that we should not stop the simulation in s_1 . If, on the other hand, we are in state \circ or if in state s_1 the guiding heuristic only picks b_1 , then we are in fact looping for more than 6 steps, and hence we stop the simulation. \triangleleft

3.4 Adapting to Games: Deflating MSECs

To extend the algorithm of [BCC+14] to SGs, instead of collapsing problematic ECs we deflate them as in [KKK18], i.e. given an MSEC, we reduce the upper bound of all states in it to the upper bound of the `bestExit` of Maximizer. In contrast to [KKK18], we cannot use the upper bound of the `bestExit` based on the true probability, but only based on our estimates. Algorithm 5 shows how to deflate an MSEC and highlights the difference, namely that we use \widehat{U} instead of U .

Algorithm 5. Black box algorithm to deflate a set of states

```

1: procedure DEFLATE(State set  $X$ )
2:   for  $s \in X$  do
3:      $U(s) = \min(U(s), \text{bestExit}(X, \widehat{U}))$ 
```

The remaining question is how to find MSECs. The approach of [KKK18] is to find MSECs by removing the suboptimal actions of Minimizer according to the current lower bound. Since it converges to the true value function, all

MSECs are eventually found [KKKW18, Lemma 2]. Since Algorithm 6 can only access the SG as a black box, there are two differences: We can only compare our estimates of the lower bound $\widehat{L}(s, a)$ to find out which actions are suboptimal. Additionally there is the problem that we might overlook an exit from an EC, and hence deflate to some value that is too small; thus we need to check that any state set FIND_MSECs returns is a δ_T -sure EC. This is illustrated in Example 3. For a bigger example of how all our functions work together, see Example 5 in [AKW19, Appendix B].

Algorithm 6. Finding MSECs in the game restricted to X for black box setting

```

1: procedure FIND_MSECs(State set  $X$ )
2:    $suboptAct_{\circlearrowleft} \leftarrow \{(s, \{a \in Av(s) \mid \widehat{L}(s, a) > L(s)\}) \mid s \in S_{\circlearrowleft} \cap X\}$ 
3:    $Av' \leftarrow Av$  without  $suboptAct_{\circlearrowleft}$ 
4:    $G' \leftarrow G$  restricted to states  $X$  and available actions  $Av'$ 
5:   return  $\{T \in MEC(G') \mid \delta_T$ -sure EC( $T$ )  $\}$ 
```

Example 3. For this example, we use the full SG from Fig. 1, including the dashed part, with $p_1, p_2 > 0$. Let $(s_0, a_1, s_1, b_2, s_2, b_1, s_1, a_2, s_2, c, 1)$ be the path generated by our simulation. Then in our partial view of the model, it seems as if $T = \{s_0, s_1\}$ is an MSEC, since using a_2 is suboptimal for the minimizing state s_0 ⁶ and according to our current knowledge a_1, b_1 and b_2 all stay inside T . If we deflated T now, all states would get an upper bound of 0, which would be incorrect.

Thus in Algorithm 6 we need to require that T is an EC δ_T -surely. This was not satisfied in the example, as the state-action pairs have not been observed the required number of times. Thus we do not deflate T , and our upper bounds stay correct. Having seen (s_1, b_2) the required number of times, we probably know that it is exiting T and hence will not make the mistake. \triangleleft

3.5 Guidance and Statistical Guarantee

It is difficult to give statistical guarantees for the algorithm we have developed so far (i.e. Algorithm 1 calling the new functions from Sects. 3.2, 3.3 and 3.4). Although we can bound the error of each function, applying them repeatedly can add up the error. Algorithm 7 shows our approach to get statistical guarantees: It interleaves a guided simulation phase (Lines 7–10) with a guaranteed standard bounded value iteration (called BVI phase) that uses our new functions (Lines 11–16).

The simulation phase builds the partial model by exploring states and remembering the counters. In the first iteration of the main loop, it chooses actions randomly. In all further iterations, it is guided by the bounds that the last BVI

⁶ For $\delta_T = 0.2$, sampling the path to target once suffices to realize that $L(s_0, a_2) > 0$.

phase computed. After \mathcal{N}_k simulations (see below for a discussion of how to choose \mathcal{N}_k), all the gathered information is used to compute one version of the partial model with probability estimates $\widehat{\mathbb{T}}$ for a certain error tolerance δ_k . We can continue with the assumption, that these probability estimates are correct, since it is only violated with a probability smaller than our error tolerance (see below for an explanation of the choice of δ_k). So in our correct partial model, we re-initialize the lower and upper bound (Line 12), and execute a guaranteed standard BVI. If the simulation phase already gathered enough data, i.e. explored the relevant states and sampled the relevant transitions often enough, this BVI achieves a precision smaller than ε in the initial state, and the algorithm terminates. Otherwise we start another simulation phase that is guided by the improved bounds.

Algorithm 7. Full algorithm for black box setting

```

1: procedure BLACKVI(SG G, target set Goal, precision  $\varepsilon > 0$ , error tolerance  $\delta > 0$ )
2:   INITIALIZE_BOUNDS
3:    $k = 1$                                       $\triangleright$  guaranteed BVI counter
4:    $\widehat{S} \leftarrow \emptyset$                     $\triangleright$  current partial model
5:   repeat
6:      $k \leftarrow 2 \cdot k$ 
7:      $\delta_k \leftarrow \frac{\delta}{k}$ 
8:     // Guided simulation phase
9:     for  $\mathcal{N}_k$  times do
10:     $X \leftarrow \text{SIMULATE}$ 
11:     $\widehat{S} \leftarrow \widehat{S} \cup X$ 
12:    INITIALIZE_BOUNDS                       $\triangleright$  Set  $\delta_{\mathbb{T}}$  as described in Section 3.2
13:    for  $k \cdot |\widehat{S}|$  times do
14:      UPDATE( $\widehat{S}$ )
15:      for  $T \in \text{FIND\_MSECs}(\widehat{S})$  do
16:        DEFFLATE( $T$ )
17:    until  $U(s_0) - L(s_0) < \varepsilon$ 

```

Choice of δ_k : For each of the full BVI phases, we construct a partial model that is correct with probability $(1 - \delta_k)$. To ensure that the sum of these errors is not larger than the specified error tolerance δ , we use the variable k , which is initialised to 1 and doubled in every iteration of the main loop. Hence for the i -th BVI, $k = 2^i$. By setting $\delta_k = \frac{\delta}{k}$, we get that $\sum_{i=1}^{\infty} \delta_k = \sum_{i=1}^{\infty} \frac{\delta}{2^i} = \delta$, and hence the error of all BVI phases does not exceed the specified error tolerance.

When to Stop Each BVI-Phase: The BVI phase might not converge if the probability estimates are not good enough. We increase the number of iterations for each BVI depending on k , because that way we ensure that it eventually is allowed to run long enough to converge. On the other hand, since we always run for finitely many iterations, we also ensure that, if we do not have enough information yet, BVI is eventually stopped. Other stopping criteria could return arbitrarily imprecise results [HM17]. We also multiply with $|\widehat{S}|$ to improve the chances of the early BVIs to converge, as that number of iterations ensures that every value has been propagated through the whole model at least once.

Discussion of the Choice of \mathcal{N}_k : The number of simulations between the guaranteed BVI phases can be chosen freely; it can be a constant number every time, or any sequence of natural numbers, possibly parameterised by e.g. k , $|\widehat{S}|$, ε or any of the parameters of G . The design of particularly efficient choices or learning mechanisms that adjust them on the fly is an interesting task left for future work. We conjecture the answer depends on the given SG and “task” that the user has for the algorithm: E.g. if one just needs a quick general estimate of the behaviour of the model, a smaller choice of \mathcal{N}_k is sensible; if on the other hand a definite precision ε certainly needs to be achieved, a larger choice of \mathcal{N}_k is required.

Theorem 1. *For any choice of sequence for \mathcal{N}_k , Algorithm 7 is an anytime algorithm with the following property: When it is stopped, it returns an interval for $V(s_0)$ that is PAC⁷ for the given error tolerance δ and some ε' , with $0 \leq \varepsilon' \leq 1$.*

Theorem 1 is the foundation of the practical usability of our algorithm. Given some time frame and some \mathcal{N}_k , it calculates an approximation for $V(s_0)$ that is probably correct. Note that the precision ε' is independent of the input parameter ε , and could in the worst case be always 1. However, practically it often is good (i.e. close to 0) as seen in the results in Sect. 4. Moreover, in our modified algorithm, we can also give a convergence guarantee as in [BCC+14]. Although mostly out of theoretical interest, in [AKW19, Appendix D.4] we design such a sequence \mathcal{N}_k , too. Since this a-priori sequence has to work in the worst case, it depends on an infeasibly large number of simulations.

Theorem 2. *There exists a choice of \mathcal{N}_k , such that Algorithm 7 is PAC for any input parameters ε, δ , i.e. it terminates almost surely and returns an interval for $V(s_0)$ of width smaller than ε that is correct with probability at least $1 - \delta$.*

⁷ Probably Approximately Correct, i.e. with probability greater than $1 - \delta$, the value lies in the returned interval of width ε' .

3.6 Utilizing the Additional Information of Grey Box Input

In this section, we consider the grey box setting, i.e. for every state-action pair (s, a) we additionally know the exact number of successors $|\text{Post}(s, a)|$. Then we can sample every state-action pair until we have seen all successors, and hence this information amounts to having qualitative information about the transitions, i.e. knowing where the transitions go, but not with which probability.

In that setting, we can improve the EC-detection and the estimated bounds in UPDATE. For EC-detection, note that the whole point of $\delta_{\mathbb{T}}$ -sure EC is to check whether there are further transitions available; in grey box, we know this and need not depend on statistics. For the bounds, note that the equations for \widehat{L} and \widehat{U} both have two parts: The usual Bellman part and the remaining probability multiplied with the most conservative guess of the bound, i.e. 0 and 1. If we know all successors of a state-action pair, we do not have to be as conservative; then we can use $\min_{t \in \text{Post}(s, a)} L(t)$ respectively $\max_{t \in \text{Post}(s, a)} U(t)$. Both these improvements have huge impact, as demonstrated in Sect. 4. However, of course, they also assume more knowledge about the model.

4 Experimental Evaluation

We implemented the approach as an extension of PRISM-Games [CFK+13a]. 11 MDPs with reachability properties were selected from the Quantitative Verification Benchmark Set [HKP+19]. Further, 4 stochastic games benchmarks from [CKJ12, SS12, CFK+13b, CKPS11] were also selected. We ran the experiments on a 40 core Intel Xeon server running at 2.20 GHz per core and having 252 GB of RAM. The tool however utilised only a single core and 1 GB of memory for the model checking. Each benchmark was ran 10 times with a timeout of 30 min. We ran two versions of Algorithm 7, one with the SG as a black box, the other as a grey box (see Definition 2). We chose $N_k = 10,000$ for all iterations. The tool stopped either when a precision of 10^{-8} was obtained or after 30 min. In total, 16 different model-property combinations were tried out. The results of the experiment are reported in Table 1.

In the black box setting, we obtained $\varepsilon < 0.1$ on 6 of the benchmarks. 5 benchmarks were ‘hard’ and the algorithm did not improve the precision below 1. For 4 of them, it did not even finish the first simulation phase. If we decrease N_k , the BVI phase is entered, but still no progress is made.

In the grey box setting, on 14 of 16 benchmarks, it took only 6 min to achieve $\varepsilon < 0.1$. For 8 these, the exact value was found within that time. Less than 50% of the state space was explored in the case of `pacman`, `pneuli-zuck-3`, `rabin-3`, `zeroconf` and `cloud-5`. A precision of $\varepsilon < 0.01$ was achieved on 15/16 benchmarks over a period of 30 min.

Table 1. Achieved precision ε' given by our algorithm in both grey and black box settings after running for a period of 30 min (See the paragraph below Theorem 1 for why we use ε' and not ε). The first set of the models are MDPs and the second set are SGs. ‘-’ indicates that the algorithm did not finish the first simulation phase and hence partial BVI was not called. m is the number of steps required by the DQL algorithm of [BCC+14] before the first update. As this number is very large, we report only $\log_{10}(m)$. For comparison, note that the age of the universe is approximately 10^{26} ns; logarithm of number of steps doable in this time is thus in the order of 26.

Model	States	Explored %	Precision		$\log_{10}(m)$
			Grey/Black	Grey	
consensus	272	100/100	0.00945	0.171	338
csma-2-2	1,038	93/93	0.00127	0.2851	1,888
firewire	83,153	55/-	0.0057	1	129,430
ij-3	7	100/100	0	0.0017	2,675
ij-10	1,023	100/100	0	0.5407	17
pacman	498	18/47	0.00058	0.0086	1,801
philosophers-3	956	56/21	0	1	2,068
pnueli-zuck-3	2,701	25/71	0	0.0285	5,844
rabin-3	27,766	7/4	0	0.026	110,097
wlan-0	2,954	100/100	0	0.8667	9,947
zeroconf	670	29/27	0.00007	0.0586	5,998
cdmsn	1,240	100/98	0	0.8588	3,807
cloud-5	8,842	49/20	0.00031	0.0487	71,484
mdsm-1	62,245	69/-	0.09625	1	182,517
mdsm-2	62,245	72/-	0.00055	1	182,517
team-form-3	12,476	64/-	0	1	54,095

Figure 2 shows the evolution of the lower and upper bounds in both the grey- and the black box settings for 4 different models. Graphs for the other models as well as more details on the results are in [AKW19, Appendix C].

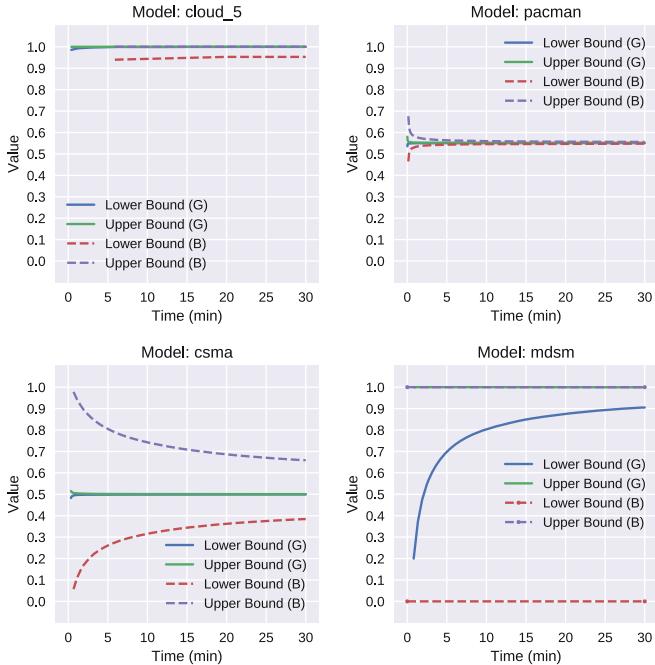


Fig. 2. Performance of our algorithm on various MDP and SG benchmarks in grey and black box settings. Solid lines denote the bounds in the grey box setting while dashed lines denote the bounds in the black box setting. The plotted bounds are obtained after each partial BVI phase, because of which they do not start at $[0, 1]$ and not at time 0. Graphs of the remaining benchmarks may be found in [AKW19, Appendix C].

5 Conclusion

We presented a PAC SMC algorithm for SG (and MDP) with the reachability objective. It is the first one for SG and the first practically applicable one. Nevertheless, there are several possible directions for further improvements. For instance, one can consider different sequences for lengths of the simulation phases, possibly also dependent on the behaviour observed so far. Further, the error tolerance could be distributed in a non-uniform way, allowing for fewer visits in rarely visited parts of end components. Since many systems are strongly connected, but at the same time feature some infrequent behaviour, this is the next bottleneck to be attacked. [KM19]

References

- [AKW19] Ashok, P., Křetínský, J.: Maximilian Weininger. PAC statistical model checking for markov decision processes and stochastic games. Technical Report [arXiv.org/abs/1905.04403](https://arxiv.org/abs/1905.04403) (2019)

- [BBB+10] Basu, A., Bensalem, S., Bozga, M., Caillaud, B., Delahaye, B., Legay, A.: Statistical abstraction and model-checking of large heterogeneous systems. In: Hatcliff, J., Zucca, E. (eds.) FMOODS/FORTE 2010. LNCS, vol. 6117, pp. 32–46. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13464-7_4
- [BCC+14] Brázdil, T., et al.: Verification of markov decision processes using learning algorithms. In: Cassez, F., Raskin, J.-F. (eds.) ATVA 2014. LNCS, vol. 8837, pp. 98–114. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11936-6_8
- [BCLS13] Boyer, B., Corre, K., Legay, A., Sedwards, S.: PLASMA-lab: a flexible, distributable statistical model checking library. In: Joshi, K., Siegle, M., Stoelinga, M., DArgenio, P.R. (eds.) QEST 2013. LNCS, vol. 8054, pp. 160–164. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40196-1_12
- [BDL+12] Bulychev, P.E., et al.: UPPAAL-SMC: statistical model checking for priced timed automata. In: QAPL (2012)
- [BFHH11] Bogdoll, J., Ferrer Fioriti, L.M., Hartmanns, A., Hermanns, H.: Partial order methods for statistical model checking and simulation. In: Bruni, R., Dingel, J. (eds.) FMOODS/FORTE 2011. LNCS, vol. 6722, pp. 59–74. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21461-5_4
- [BHH12] Bogdoll, J., Hartmanns, A., Hermanns, H.: Simulation and statistical model checking for modestly nondeterministic models. In: Schmitt, J.B. (ed.) MMB&DFT 2012. LNCS, vol. 7201, pp. 249–252. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28540-0_20
- [BK08] Baier, C., Katoen, J.-P.: Principles of Model Checking. MIT Press (2008). ISBN 978-0-262-02649-9
- [BT99] Brafman, R.I., Tennenholtz, M.: A near-optimal poly-time algorithm for learning a class of stochastic games. In: IJCAI, pp. 734–739 (1999)
- [CFK+13a] Chen, T., Forejt, V., Kwiatkowska, M., Parker, D., Simaitis, A.: PRISM-games: a model checker for stochastic multi-player games. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 185–191. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_13
- [CFK+13b] Chen, T., Forejt, V., Kwiatkowska, M., Parker, D., Simaitis, A.: Automatic verification of competitive stochastic systems. Formal Meth. Syst. Des. **43**(1), 61–92 (2013)
- [CH08] Chatterjee, K., Henzinger, T.A.: Value iteration. In: Grumberg, O., Veith, H. (eds.) 25 Years of Model Checking. LNCS, vol. 5000, pp. 107–138. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-69850-0_7
- [CH12] Chatterjee, K., Henzinger, T.A.: A survey of stochastic ω -regular games. J. Comput. Syst. Sci. **78**(2), 394–413 (2012)
- [CKJ12] Calinescu, R., Kikuchi, S., Johnson, K.: Compositional reverification of probabilistic safety properties for large-scale complex IT systems. In: Calinescu, R., Garlan, D. (eds.) Monterey Workshop 2012. LNCS, vol. 7539, pp. 303–329. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34059-8_16
- [CKPS11] Chen, T., Kwiatkowska, M., Parker, D., Simaitis, A.: Verifying team formation protocols with probabilistic model checking. In: Leite, J., Torroni,

- P., Ågotnes, T., Boella, G., van der Torre, L. (eds.) CLIMA 2011. LNCS (LNNAI), vol. 6814, pp. 190–207. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22359-4_14
- [Con92] Condon, A.: The complexity of stochastic games. *Inf. Comput.* **96**(2), 203–224 (1992)
- [CZ11] Clarke, E.M., Zuliani, P.: Statistical model checking for cyber-physical systems. In: ATVA, pp. 1–12 (2011)
- [DDL+12] David, A., et al.: Statistical model checking for stochastic hybrid systems. In: HSB, pp. 122–136 (2012)
- [DDL+13] David, A., Du, D., Guldstrand Larsen, K., Legay, A., Mikućionis, M.: Optimizing control strategy using statistical model checking. In: Brat, G., Rungeta, N., Venet, A. (eds.) NFM 2013. LNCS, vol. 7871, pp. 352–367. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38088-4_24
- [DHKP16] Daca, P., Henzinger, T.A., Kretínský, J., Petrov, T.: Faster statistical model checking for unbounded temporal properties. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 112–129. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_7
- [DHS18] D’Argenio, P.R., Hartmanns, A., Sedwards, S.: Lightweight statistical model checking in nondeterministic continuous time. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11245, pp. 336–353. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03421-4_22
- [DLL+11a] David, A., et al.: Statistical model checking for networks of priced timed automata. In: Fahrenberg, U., Tripakis, S. (eds.) FORMATS 2011. LNCS, vol. 6919, pp. 80–96. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24310-3_7
- [DLL+11b] David, A., Larsen, K.G., Legay, A., Mikućionis, M., Wang, Z.: Time for statistical model checking of real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 349–355. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_27
- [DLST15] D’Argenio, P., Legay, A., Sedwards, S., Traonouez, L.-M.: Smart sampling for lightweight verification of markov decision processes. *STTT* **17**(4), 469–484 (2015)
- [EGF12] Ellen, C., Gerwinn, S., Fränzle, M.: Confidence bounds for statistical model checking of probabilistic hybrid systems. In: Jurdziński, M., Ničković, D. (eds.) FORMATS 2012. LNCS, vol. 7595, pp. 123–138. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33365-1_10
- [FT14] Fu, J., Topcu, U.: Probably approximately correct MDP learning and control with temporal logic constraints. In: Robotics: Science and Systems (2014)
- [HAK18] Hasanbeig, M., Abate, A., Kroening, D.: Logically-correct reinforcement learning. CoRR, 1801.08099 (2018)
- [HAK19] Hasanbeig, M., Abate, A., Kroening, D.: Certified reinforcement learning with logic guidance. CoRR, abs/1902.00778 (2019)
- [HJB+10] He, R., Jennings, P., Basu, S., Ghosh, A.P., Wu, H.: A bounded statistical approach for model checking of unbounded until properties. In: ASE, pp. 225–234 (2010)

- [HKP+19] Hartmanns, A., Klauck, M., Parker, D., Quatmann, T., Ruijters, E.: The quantitative verification benchmark set. In: TACAS 2019 (2019, to appear)
- [HLMP04] Héault, T., Lassaigne, R., Magniette, F., Peyronnet, S.: Approximate probabilistic model checking. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 73–84. Springer, Heidelberg (2004). <https://doi.org/10.1007/978-3-540-24622-8>
- [HM17] Haddad, S., Monmege, B.: Interval iteration algorithm for MDPs and IMDPs. Theor. Comput. Sci. (2017)
- [HMZ+12] Henriques, D., Martins, J., Zuliani, P., Platzer, A., Clarke, E.M.: Statistical model checking for Markov decision processes. In: QEST, pp. 84–93 (2012)
- [HPS+19] Hahn, E.M., Perez, M., Schewe, S., Somenzi, F., Trivedi, A., Wojtczak, D.: Omega-regular objectives in model-free reinforcement learning. In: Vojnar, T., Zhang, L. (eds.) TACAS 2019. LNCS, vol. 11427, pp. 395–412. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17462-0_27
- [JCL+09] Jha, S.K., Clarke, E.M., Langmead, C.J., Legay, A., Platzer, A., Zuliani, P.: A bayesian approach to model checking biological systems. In: Degano, P., Gorrieri, R. (eds.) CMSB 2009. LNCS, vol. 5688, pp. 218–234. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03845-7_15
- [JLS12] Jegourel, C., Legay, A., Sedwards, S.: A platform for high performance statistical model checking – PLASMA. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 498–503. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28756-5_37
- [KKKW18] Kelmendi, E., Krämer, J., Křetínský, J., Weininger, M.: Value iteration for simple stochastic games: stopping criterion and learning algorithm. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 623–642. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_36
- [KM19] Křetínský, J., Meggendorfer, T.: Of cores: a partial-exploration framework for Markov decision processes. Submitted 2019
- [KNP11] Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_47
- [Lar12] Larsen, K.G.: Statistical model checking, refinement checking, optimization, for stochastic hybrid systems. In: Jurdziński, M., Ničković, D. (eds.) FORMATS 2012. LNCS, vol. 7595, pp. 7–10. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33365-1_2
- [Lar13] Guldstrand Larsen, K.: Priced timed automata and statistical model checking. In: Johnsen, E.B., Petre, L. (eds.) IFM 2013. LNCS, vol. 7940, pp. 154–161. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38613-8_11
- [Lit94] Littman, M.L.: Markov games as a framework for multi-agent reinforcement learning. In: ICML, pp. 157–163 (1994)
- [LN81] Lakshmivarahan, S., Narendra, K.S.: Learning algorithms for two-person zero-sum stochastic games with incomplete information. Math. Oper. Res. **6**(3), 379–386 (1981)

- [LP08] Lassaigne, R., Peyronnet, S.: Probabilistic verification and approximation. *Ann. Pure Appl. Logic* **152**(1–3), 122–131 (2008)
- [LP12] Lassaigne, R., Peyronnet, S.: Approximate planning and verification for large Markov decision processes. In: SAC, pp. 1314–1319, (2012)
- [LST14] Legay, A., Sedwards, S., Traonouez, L.-M.: Scalable verification of markov decision processes. In: Canal, C., Idani, A. (eds.) SEFM 2014. LNCS, vol. 8938, pp. 350–362. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-15201-1_23
- [Mar75] Martin, D.A.: Borel determinacy. *Ann. Math.* **102**(2), 363–371 (1975)
- [MLG05] McMahan, H.B., Likhachev, M., Gordon, G.J.: Bounded real-time dynamic programming: RTDP with monotone upper bounds and performance guarantees. In: In ICML 2005, pp. 569–576 (2005)
- [Nor98] Norris, J.R.: Markov Chains. Cambridge University Press, Cambridge (1998)
- [PGL+13] Palaniappan, S.K., Gyori, B.M., Liu, B., Hsu, D., Thiagarajan, P.S.: Statistical model checking based calibration and analysis of bio-pathway models. In: Gupta, A., Henzinger, T.A. (eds.) CMSB 2013. LNCS, vol. 8130, pp. 120–134. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40708-6_10
- [Put14] Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. Wiley, Hoboken (2014)
- [RF91] Raghavan, T.E.S., Filar, J.A.: Algorithms for stochastic games – a survey. *Z. Oper. Res.* **35**(6), 437–472 (1991)
- [RP09] El Rabih, D., Pekergin, N.: Statistical model checking using perfect simulation. In: Liu, Z., Ravn, A.P. (eds.) ATVA 2009. LNCS, vol. 5799, pp. 120–134. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04761-9_11
- [SB98] Sutton, R., Barto, A.: Reinforcement Learning: An Introduction. MIT Press, Cambridge (1998)
- [SKC+14] Sadigh, D., Kim, E.S., Coogan, S., Sastry, S.S.S., Sanjit, A.: A learning based approach to control synthesis of markov decision processes for linear temporal logic specifications. In: CDC, pp. 1091–1096 (2014)
- [SLW+06] Strehl, A.L., Li, L., Wiewiora, E., Langford, J., Littman, M.L.: PAC model-free reinforcement learning. In: ICML, pp. 881–888 (2006)
- [SS12] Saffre, F., Simaitis, A.: Host selection through collective decision. *ACM Trans. Auton. Adapt. Syst.* **7**(1), 4:1–4:16 (2012)
- [SVA04] Sen, K., Viswanathan, M., Agha, G.: Statistical model checking of black-box probabilistic systems. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 202–215. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27813-9_16
- [SVA05] Sen, K., Viswanathan, M., Agha, G.: On statistical model checking of stochastic systems. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 266–280. Springer, Heidelberg (2005). https://doi.org/10.1007/11513988_26
- [WT16] Wen, M., Topcu, U.: Probably approximately correct learning in stochastic games with temporal logic specifications. In: IJCAI, pp. 3630–3636 (2016)
- [YCZ10] Younes, H.L.S., Clarke, E.M., Zuliani, P.: Statistical verification of probabilistic properties with unbounded until. In: Davies, J., Silva, L., Simao, A. (eds.) SBMF 2010. LNCS, vol. 6527, pp. 144–160. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19829-8_10

- [YKNP06] Younes, H.L.S., Kwiatkowska, M.Z., Norman, G., Parker, D.: Numerical vs. statistical probabilistic model checking. *STTT* **8**(3), 216–228 (2006)
- [YS02a] Younes, H.L.S., Simmons, R.G.: Probabilistic verification of discrete event systems using acceptance sampling. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 223–235. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45657-0_17
- [ZPC10] Zuliani, P., Platzer, A., Clarke, E.M.: Bayesian statistical model checking with application to simulink/stateflow verification. In: HSCC, pp. 243–252 (2010)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Symbolic Monitoring Against Specifications Parametric in Time and Data

Masaki Waga^{1,2,3}(✉) , Étienne André^{1,4,5} ,
and Ichiro Hasuo^{1,2}



¹ National Institute of Informatics, Tokyo, Japan
mwaga@nii.ac.jp

² SOKENDAI (The Graduate University for Advanced Studies), Tokyo, Japan
³ JSPS Research Fellow, Tokyo, Japan

⁴ Université Paris 13, LIPN, CNRS, UMR 7030, 93430 Villetaneuse, France

⁵ JFLI, CNRS, Tokyo, Japan

Abstract. Monitoring consists in deciding whether a log meets a given specification. In this work, we propose an automata-based formalism to monitor logs in the form of actions associated with time stamps and arbitrarily data values over infinite domains. Our formalism uses both timing parameters and data parameters, and is able to output answers symbolic in these parameters and in the log segments where the property is satisfied or violated. We implemented our approach in an ad-hoc prototype SYMON, and experiments show that its high expressive power still allows for efficient online monitoring.

1 Introduction

Monitoring consists in checking whether a sequence of data (a log or a signal) satisfies or violates a specification expressed using some formalism. Offline monitoring consists in performing this analysis after the system execution, as the technique has access to the entire log in order to decide whether the specification is violated. In contrast, online monitoring can make a decision earlier, ideally as soon as a witness of the violation of the specification is encountered.

Using existing formalisms (e.g., the metric first order temporal logic [14]), one can check whether a given bank customer withdraws more than 1,000 € every week. With formalisms extended with data, one may even *identify* such customers. Or, using an extension of the signal temporal logic (STL) [18], one can ask: “is that true that the value of variable x is always copied to y exactly 4 time units later?” However, questions relating time and data using parameters become

This work is partially supported by JST ERATO HASUO Metamathematics for Systems Design Project (No. JPMJER1603), by JSPS Grants-in-Aid No. 15KT0012 & 18J22498 and by the ANR national research program PACS (ANR-14-CE28-0002).

much harder (or even impossible) to express using existing formalisms: “what are the users and time frames during which a user withdraws more than half of the total bank withdrawals within seven days?” And even, can we *synthesize* the durations (not necessarily 7 days) for which this specification holds? Or “what is the set of variables for which there exists a duration within which their value is always copied to another variable?” In addition, detecting periodic behaviors without knowing the period can be hard to achieve using existing formalisms.

In this work, we address the challenging problem to monitor logs enriched with both timing information and (infinite domain) data. In addition, we significantly push the existing limits of expressiveness so as to allow for a further level of abstraction using *parameters*: our specification can be both parametric in the *time* and in the *data*. The answer to this symbolic monitoring is richer than a pure Boolean answer, as it *synthesizes* the values of both time and data parameters for which the specification holds. This allows us notably to detect periodic behaviors without knowing the period while being symbolic in terms of data. For example, we can *synthesize* *variable names* (data) and *delays* for which variables will have their value copied to another data within the aforementioned delay. In addition, we show that we can detect the log *segments* (start and end date) for which a specification holds.

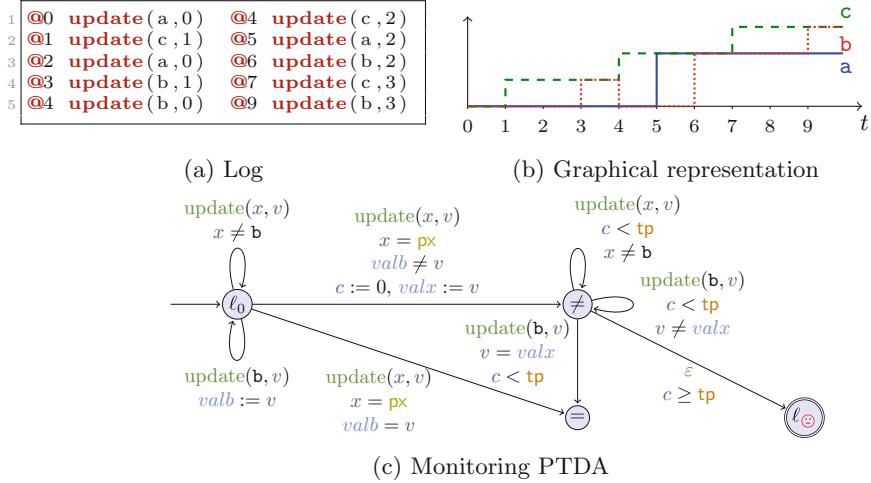
Example 1. Consider a system updating three variables a , b and c (i.e., strings) to values (rationals). An example of log is given in Fig. 1a. Although our work is event-based, we can give a graphical representation similar to that of signals in Fig. 1b. Consider the following property: “for any variable px , whenever an update of that variable occurs, then within strictly less than tp time units, the value of variable b must be equal to that update”. The *variable parameter* px is compared with string values and the *timing parameter* tp is used in the timing constraints. We are interested in checking for which values of px and tp this property is violated. This can be seen as a synthesis problem in both the variable and timing parameters. For example, $\text{px} = c$ and $\text{tp} = 1.5$ is a violation of the specification, as the update of c to 2 at time 4 is not propagated to b within 1.5 time unit. Our algorithm outputs such violation by a constraint e.g., $\text{px} = c \wedge \text{tp} \leq 2$. In contrast, the value of any signal at any time is always such that either b is equal to that signal, or the value of b will be equal to that value within at most 2 time units. Thus, the specification holds for any valuation of the variable parameter px , provided $\text{tp} > 2$.

We propose an automata-based approach to perform monitoring parametric in both time and data. We implement our work in a prototype SYMON and perform experiments showing that, while our formalism allows for high expressiveness, it is also tractable even for online monitoring.

We believe our framework balances expressiveness and monitoring performance well: (i) Regarding expressiveness, comparison with the existing work is summarized in Table 1 (see Sect. 2 for further details). (ii) Our monitoring is *complete*, in the sense that it returns a symbolic constraint characterizing *all* the parameter valuations that match a given specification. (iii) We also achieve

Table 1. Comparison of monitoring expressiveness

Work	[7]	[18]	[14]	[13]	[30]	[26]	[4]	[9]	This work
Timing parameters	✓	✗	?	?	?	✗	✓	✗	✓
Data	✓	✓	✓	✓	✓	✓	✗	✓	✓
Parametric data	✓	✗	✓	✓	✓	✓	✗	✓	✓
Memory	✗	✓	✓	✓	✓	✓	✗	✗	✓
Aggregation	✗	✗	✗	✓	✓	✓	✗	✗	✓
Complete parameter identification	✓	N/A	✓/✓	✓/✓	N/A	N/A	✓/✓	✓/✓	✓

**Fig. 1.** Monitoring copy to b within tp time units

reasonable monitoring speed, especially given the degree of parametrization in our formalism. Note that it is not easy to formally claim superiority in expressiveness: proofs would require arguments such as the pumping lemma; and such formal comparison does not seem to be a concern of the existing work. Moreover, such formal comparison bears little importance for industrial practitioners: expressivity via an elaborate encoding is hardly of practical use. We also note that, in the existing work, we often observe gaps between the formalism in a theory and the formalism that the resulting tool actually accepts. This is not the case with the current framework.

Outline. After discussing related works in Sect. 2, we introduce the necessary preliminaries in Sect. 3, and our parametric timed data automata in Sect. 4. We present our symbolic monitoring approach in Sect. 5 and conduct experiments in Sect. 6. We conclude in Sect. 7.

2 Related Works

Robustness and Monitoring. Robust (or quantitative) monitoring extends the binary question whether a log satisfies a specification by asking “by how much” the specification is satisfied. The quantification of the distance between a signal and a signal temporal logic (STL) specification has been addressed in, e.g., [20–23, 25, 27] (or in a slightly different setting in [5]). The distance can be understood in terms of space (“signals”) or time. In [6], the distance also copes for reordering of events. In [10], the *robust pattern matching problem* is considered over signal regular expressions, by quantifying the distance between the signal regular expression specification and the *segments* of the signal. For piecewise-constant and piecewise-linear signals, the problem can be effectively solved using a finite union of convex polyhedra. While our framework does not fit in robust monitoring, we can simulate both the robustness w.r.t. time (using timing parameters) and w.r.t. data, e.g., signal values (using data parameters).

Monitoring with Data. The tool MARQ [30] performs monitoring using Quantified Event Automata (QEA) [12]. This approach and ours share the automata-based framework, the ability to express some first-order properties using “events containing data” (which we encode using local variables associated with actions), and data may be quantified. However, [30] does not seem to natively support specification parametric in time; in addition, [30] does not perform complete (“symbolic”) parameters synthesis, but outputs the violating entries of the log.

The metric first order temporal logic (MFOTL) allows for a high expressiveness by allowing universal and existential quantification over data—which can be seen as a way to express parameters. A monitoring algorithm is presented for a safety fragment of MFOTL in [14]. Aggregation operators are added in [13], allowing to compute *sums* or *maximums* over data. A fragment of this logics is implemented in MONPOLY [15]. While these works are highly expressive, they do not natively consider timing parameters; in addition, MONPOLY does not output symbolic answers, i. e., symbolic conditions on the parameters to ensure validity of the formula.

In [26], binary decision diagrams (BDDs) are used to symbolically represent the observed data in QTL. This can be seen as monitoring data against a parametric specification, with a symbolic internal encoding. However, their implementation DEJAVU only outputs *concrete* answers. In contrast, we are able to provide symbolic answers (both in timing and data parameters), e.g., in the form of union of polyhedra for rationals, and unions of string constraints using equalities (=) and inequalities (\neq).

Freeze Operator. In [18], STL is extended with a freeze operator that can “remember” the value of a signal, to compare it to a later value of the same signal. This logic STL^* can express properties such as “In the initial 10 s, x copies the values of y within a delay of 4 s”: $\mathbf{G}_{[0,10]} * (\mathbf{G}_{[0,4]} y^* = x)$. While the setting is somehow different (STL^* operates over signals while we operate over timed data words), the requirements such as the one above can easily be encoded

in our framework. In addition, we are able to *synthesize* the delay within which the values are always copied, as in Example 1. In contrast, it is not possible to determine using STL* which variables and which delays violate the specification.

Monitoring with Parameters. In [7], a log in the form of a dense-time real-valued signal is tested against a parameterized extension of STL, where parameters can be used to model uncertainty both in signal values and in timing values. The output comes in the form of a subset of the parameters space for which the formula holds on the log. In [9], the focus is only on signal parameters, with an improved efficiency by reusing techniques from the *robust* monitoring. Whereas [7, 9] fit in the framework of signals and temporal logics while we fit in words and automata, our work shares similarities with [7, 9] in the sense that we can express data parameters; in addition, [9] is able as in our work to exhibit the segment of the log associated with the parameters valuations for which the specification holds. A main difference however is that we can use memory and aggregation, thanks to arithmetic on variables.

In [24], the problem of *inferring* temporal logic formulae with constraints that hold in a given numerical data time series is addressed.

Timed Pattern Matching. A recent line of work is that of timed pattern matching, that takes as input a log and a specification, and decides *where* in the log the specification is satisfied or violated. On the one hand, a line of works considers signals, with specifications either in the form of timed regular expressions [11, 31–33], or a temporal logic [34]. On the other hand, a line of works considers timed words, with specifications in the form of timed automata [4, 36]. We will see that our work can also encode parametric timed pattern matching. Therefore, our work can be seen as a two-dimensional extension of both lines of works: first, we add timing parameters ([4] also considers similar timing parameters) and, second, we add data—themselves extended with parameters. That is, coming back to Example 1, [31–33, 36] could only infer the segments of the log for which the property is violated for a given (fixed) variable and a given (fixed) timing parameter; while [4] could infer both the segments of the log and the timing parameter valuations, but not which variable violates the specification.

Summary. We compare related works in Table 1. “Timing parameters” denote the ability to synthesize unknown constants used in timing constraints (e.g., modalities intervals, or clock constraints). “?” denotes works not natively supporting this, although it might be encoded. The term “Data” refers to the ability to manage logs over infinite domains (apart from timestamps). For example, the log in Fig. 1a features, beyond timestamps, both string (variable name) and rationals (value). Also, works based on real-valued signals are naturally able to manage (at least one type of) data. “Parametric data” refer to the ability to express formulas where data (including signal values) are compared to (quantified or unquantified) variables or unknown parameters; for example, in the log in Fig. 1a, an example of property parametric in data is to synthesize the parameters for which the difference of values between two consecutive updates of

variable px is always below pv , where px is a string parameter and pv a rational-valued parameter. “Memory” is the ability to remember *past* data; this can be achieved using e.g., the freeze operator of STL*, or variables (e.g., in [14, 26, 30]). “Aggregation” is the ability to aggregate data using operators such as sum or maximum; this allows to express properties such as “A user must not withdraw more than \$10,000 within a 31 day period” [13]. This can be supported using dedicated aggregation operators [13] or using variables ([30], and our work). “Complete parameter identification” denotes the *synthesis* of the set of parameters that satisfy or violate the property. Here, “N/A” denotes the absence of parameter [18], or when parameters are used in a way (existentially or universally quantified) such as the identification is not explicit (instead, the position of the log where the property is violated is returned [26]). In contrast, we return in a *symbolic* manner (as in [4, 7]) the exact set of (data and timing) parameters for which a property is satisfied. “ \checkmark/\times ” denotes “yes” in the theory paper, but not in the tool.

3 Preliminaries

Clocks, Timing Parameters and Timed Guards. We assume a set $\mathbb{C} = \{c_1, \dots, c_H\}$ of *clocks*, i.e., real-valued variables that evolve at the same rate. A *clock valuation* is $\nu : \mathbb{C} \rightarrow \mathbb{R}_{\geq 0}$. We write $\mathbf{0}$ for the clock valuation assigning 0 to all clocks. Given $d \in \mathbb{R}_{\geq 0}$, $\nu + d$ is s.t. $(\nu + d)(c) = \nu(c) + d$, for all $c \in \mathbb{C}$. Given $R \subseteq \mathbb{C}$, we define the *reset* of a valuation ν , denoted by $[\nu]_R$, as follows: $[\nu]_R(c) = 0$ if $c \in R$, and $[\nu]_R(c) = \nu(c)$ otherwise.

We assume a set $\mathbb{TP} = \{\text{tp}_1, \dots, \text{tp}_J\}$ of *timing parameters*. A *timing parameter valuation* is $\gamma : \mathbb{TP} \rightarrow \mathbb{Q}_+$. We assume $\bowtie \in \{<, \leq, =, \geq, >\}$. A *timed guard* tg is a constraint over $\mathbb{C} \cup \mathbb{TP}$ defined by a conjunction of inequalities of the form $c \bowtie d$, or $c \bowtie \text{tp}$ with $d \in \mathbb{N}$ and $\text{tp} \in \mathbb{TP}$. Given tg , we write $\nu \models \gamma(tg)$ if the expression obtained by replacing each c with $\nu(c)$ and each tp with $\gamma(\text{tp})$ in tg evaluates to true.

Variables, Data Parameters and Data Guards. For sake of simplicity, we assume a *single* infinite domain \mathbb{D} for data. The formalism defined in Sect. 4 can be extended in a straightforward manner to different domains for different variables (and our implementation does allow for different types). The case of *finite* data domain is immediate too. We define this formalism in an *abstract* manner, so as to allow a sort of parameterized domain.

We assume a set $\mathbb{V} = \{v_1, \dots, v_M\}$ of *variables* valued over \mathbb{D} . These variables are internal variables, that allow an high expressive power in our framework, as they can be compared or updated to other variables or parameters. We also assume a set $\mathbb{LV} = \{lv_1, \dots, lv_O\}$ of *local variables* valued over \mathbb{D} . These variables will only be used locally along a transition in the “argument” of the action (e.g., x and v in `update(x, v)`), and in the associated guard and (right-hand part of) updates. We assume a set $\mathbb{VP} = \{\text{vp}_1, \dots, \text{vp}_N\}$ of *data parameters*, i.e., unknown variable constants.

A *data type* $(\mathbb{D}, \mathcal{DE}, \mathcal{DU})$ is made of (i) an infinite domain \mathbb{D} , (ii) a set of admissible Boolean expressions \mathcal{DE} (that may rely on \mathbb{V} , \mathbb{LV} and \mathbb{VP}), which will define the type of guards over variables in our subsequent automata, and (iii) a domain for updates \mathcal{DU} (that may rely on \mathbb{V} , \mathbb{LV} and \mathbb{VP}), which will define the type of updates of variables in our subsequent automata.

Example 2. As a first example, let us define the data type for rationals. We have $\mathbb{D} = \mathbb{Q}$. Let us define Boolean expressions. A *rational comparison* is a constraint over $\mathbb{V} \cup \mathbb{LV} \cup \mathbb{VP}$ defined by a conjunction of inequalities of the form $v \bowtie d$, $v \bowtie v'$, or $v \bowtie vp$ with $v, v' \in \mathbb{V} \cup \mathbb{LV}$, $d \in \mathbb{Q}$ and $vp \in \mathbb{VP}$. \mathcal{DE} is the set of all rational comparisons over $\mathbb{V} \cup \mathbb{LV} \cup \mathbb{VP}$. Let us then define updates. First, a linear arithmetic expression over $\mathbb{V} \cup \mathbb{LV} \cup \mathbb{VP}$ is $\sum_i \alpha_i v_i + \beta$, where $v_i \in \mathbb{V} \cup \mathbb{LV} \cup \mathbb{VP}$ and $\alpha_i, \beta \in \mathbb{Q}$. Let $\mathcal{LA}(\mathbb{V} \cup \mathbb{LV} \cup \mathbb{VP})$ denote the set of arithmetic expressions over \mathbb{V} , \mathbb{LV} and \mathbb{VP} . We then have $\mathcal{DU} = \mathcal{LA}(\mathbb{V} \cup \mathbb{LV} \cup \mathbb{VP})$.

As a second example, let us define the data type for strings. We have $\mathbb{D} = \mathbb{S}$, where \mathbb{S} denotes the set of all strings. A *string comparison* is a constraint over $\mathbb{V} \cup \mathbb{LV} \cup \mathbb{VP}$ defined by a conjunction of comparisons of the form $v \approx s$, $v \approx v'$, or $v \approx vp$ with $v, v' \in \mathbb{V} \cup \mathbb{LV}$, $s \in \mathbb{S}$, $vp \in \mathbb{VP}$ and $\approx \in \{=, \neq\}$. \mathcal{DE} is the set of all string comparisons over $\mathbb{V} \cup \mathbb{LV} \cup \mathbb{VP}$. $\mathcal{DU} = \mathbb{V} \cup \mathbb{LV} \cup \mathbb{S}$, i.e., a string variable can be assigned another string variable, or a concrete string.

A *variable valuation* is $\mu : \mathbb{V} \rightarrow \mathbb{D}$. A *local variable valuation* is a partial function $\eta : \mathbb{LV} \nrightarrow \mathbb{D}$. A *data parameter valuation* is $\zeta : \mathbb{VP} \rightarrow \mathbb{D}$. Given a data guard $dg \in \mathcal{DE}$, a variable valuation μ , a local variable valuation η defined for the local variables in dg , and a data parameter valuation ζ , we write $(\mu, \eta) \models \zeta(dg)$ if the expression obtained by replacing within dg all occurrences of each data parameter vp_i by $\zeta(vp_i)$ and all occurrences of each variable v_j (resp. local variable lv_k) with its concrete valuation $\mu(v_j)$ (resp. $\eta(lv_k)$) evaluates to true.

A parametric data update is a partial function $PDU : \mathbb{V} \nrightarrow \mathcal{DU}$. That is, we can assign to a variable an expression over data parameters and other variables, according to the data type. Given a parametric data update PDU , a variable valuation μ , a local variable valuation η (defined for all local variables appearing in PDU), and a data parameter valuation ζ , we define $[\mu]_{\eta(\zeta(PDU))} : \mathbb{V} \rightarrow \mathbb{D}$ as:

$$[\mu]_{\eta(\zeta(PDU))}(v) = \begin{cases} \mu(v) & \text{if } PDU(v) \text{ is undefined} \\ \eta(\mu(\zeta(PDU(v)))) & \text{otherwise} \end{cases}$$

where $\eta(\mu(\zeta(PDU(v))))$ denotes the replacement within the update expression $PDU(v)$ of all occurrences of each data parameter vp_i by $\zeta(vp_i)$, and all occur-

Table 2. Variables, parameters and valuations used in guards

	Timed guards		Data guards		
	Clock	Timing parameter	(Data) variable	Local variable	Data parameter
Variable	c	tp	v	lv	vp
Valuation	ν	γ	μ	η	ζ

rences of each variable v_j (resp. local variable lv_k) with its concrete valuation $\mu(v_j)$ (resp. $\eta(lv_k)$). Observe that this replacement gives a value in \mathbb{D} , therefore the result of $[\mu]_{\eta(\zeta(\text{PDU}))}$ is indeed a data parameter valuation $\mathbb{V} \rightarrow \mathbb{D}$. That is, $[\mu]_{\eta(\zeta(\text{PDU}))}$ computes the new (non-parametric) variable valuation obtained after applying to μ the partial function PDU valued with ζ .

Example 3. Consider the data type for rationals, the variables set $\{v_1, v_2\}$, the local variables set $\{lv_1, lv_2\}$ and the parameters set $\{\text{vp}_1\}$. Let μ be the variable valuation such that $\mu(v_1) = 1$ and $\mu(v_2) = 2$, and η be the local variable valuation such that $\eta(lv_1) = 2$ and $\eta(lv_2)$ is not defined. Let ζ be the data parameter valuation such that $\zeta(\text{vp}_1) = 1$. Consider the parametric data update function PDU such that $\text{PDU}(v_1) = 2 \times v_1 + v_2 - lv_1 + \text{vp}_1$, and $\text{PDU}(v_2)$ is undefined. Then the result of $[\mu]_{\eta(\zeta(\text{PDU}))}$ is μ' such that $\mu'(v_1) = 2 \times \mu(v_1) + \mu(v_2) - \eta(lv_1) + \zeta(\text{vp}_1) = 3$ and $\mu'(v_2) = 2$.

4 Parametric Timed Data Automata

We introduce here Parametric timed data automata (PTDAs). They can be seen as an extension of parametric timed automata [2] (that extend timed automata [1] with parameters in place of integer constants) with unbounded data variables and parametric variables. PTDAs can also be seen as an extension of some extensions of timed automata with data (see e.g., [16, 19, 29]), that we again extend with both data parameters and timing parameters. Or as an extension of quantified event automata [12] with explicit time representation using clocks, and further augmented with timing parameters. PTDAs feature both timed guards and data guards; we summarize the various variables and parameters types together with their notations in Table 2.

We will associate local variables with actions (which can be seen as *predicates*). Let $\text{Dom} : \Sigma \rightarrow 2^{\mathbb{L}\mathbb{V}}$ denote the set of local variables associated with each action. Let $\text{Var}(dg)$ (resp. $\text{Var}(\text{PDU})$) denote the set of variables occurring in dg (resp. PDU).

Definition 1 (PTDA). Given a data type $(\mathbb{D}, \mathcal{DE}, \mathcal{DU})$, a parametric timed data automaton (PTDA) \mathcal{A} over this data type is a tuple $\mathcal{A} = (\Sigma, L, \ell_0, F, \mathbb{C}, \mathbb{TP}, \mathbb{V}, \mathbb{L}\mathbb{V}, \mu_0, \mathbb{VP}, E)$, where:

1. Σ is a finite set of actions,
2. L is a finite set of locations, $\ell_0 \in L$ is the initial location,
3. $F \subseteq L$ is the set of accepting locations,
4. \mathbb{C} is a finite set of clocks,
5. \mathbb{TP} is a finite set of timing parameters,
6. \mathbb{V} (resp. $\mathbb{L}\mathbb{V}$) is a finite set of variables (resp. local variables) over \mathbb{D} ,
7. μ_0 is the initial variable valuation,
8. \mathbb{VP} is a finite set of data parameters,

9. E is a finite set of edges $e = (\ell, tg, dg, a, R, \text{PDU}, \ell')$ where (i) $\ell, \ell' \in L$ are the source and target locations, (ii) tg is a timed guard, (iii) $dg \in \mathcal{DE}$ is a data guard such as $\text{Var}(dg) \cap \mathbb{LV} \subseteq \text{Dom}(a)$, (iv) $a \in \Sigma$, (v) $R \subseteq \mathbb{C}$ is a set of clocks to be reset, and (vi) $\text{PDU} : \mathbb{V} \rightarrow \mathcal{DU}$ is the parametric data update function such that $\text{Var}(\text{PDU}) \cap \mathbb{LV} \subseteq \text{Dom}(a)$.

The domain conditions on dg and PDU ensure that the local variables used in the guard (resp. update) are only those in the action signature $\text{Dom}(a)$.

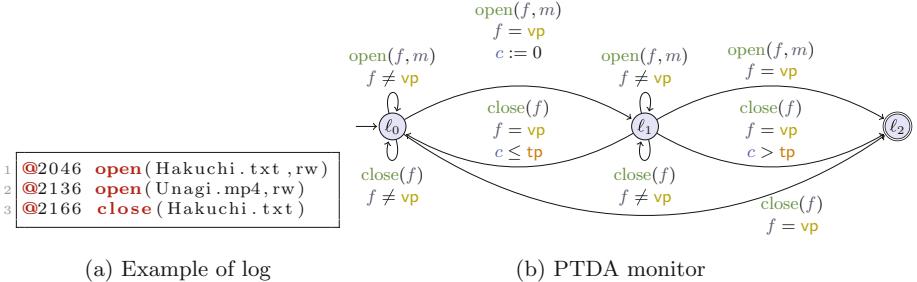


Fig. 2. Monitoring proper file opening and closing

Example 4. Consider the PTDA in Fig. 2b over the data type for strings. We have $\mathbb{C} = \{c\}$, $\mathbb{TP} = \{\text{tp}\}$, $\mathbb{V} = \emptyset$ and $\mathbb{LV} = \{f, m\}$. $\text{Dom}(\text{open}) = \{f, m\}$ while $\text{Dom}(\text{close}) = \{f\}$. ℓ_2 is the only accepting location, modeling the violation of the specification.

This PTDA (freely inspired by a formula from [26] further extended with timing parameters) monitors the improper file opening and closing, i. e., a file already open should not be open again, and a file that is open should not be closed too late. The data parameter vp is used to *symbolically* monitor a given file name, i. e., we are interested in opening and closings of this file only, while other files are disregarded (specified using the self-loops in ℓ_0 and ℓ_1 with data guard $f \neq \text{vp}$). Whenever f is opened (transition from ℓ_0 to ℓ_1), a clock c is reset. Then, in ℓ_1 , if f is closed within tp time units (timed guard “ $c \leq \text{tp}$ ”), then the system goes back to ℓ_0 . However, if instead f is opened again, this is an incorrect behavior and the system enters ℓ_2 via the upper transition. The same occurs if f is closed more than tp time units after opening.

Given a data parameter valuation ζ and a timing parameter valuation γ , we denote by $\gamma|\zeta(\mathcal{A})$ the resulting *timed data automaton (TDA)*, i. e., the non-parametric structure where all occurrences of a parameter vp_i (resp. tp_j) have been replaced by $\zeta(\text{vp}_i)$ (resp. $\gamma(\text{tp}_j)$). Note that, if $\mathbb{V} = \mathbb{LV} = \emptyset$, then \mathcal{A} is a *parametric timed automaton* [2] and $\gamma|\zeta(\mathcal{A})$ is a *timed automaton* [1].

We now equip our TDAs with a concrete semantics.

Definition 2 (Semantics of a TDA). Given a PTDA $\mathcal{A} = (\Sigma, L, \ell_0, F, \mathbb{C}, \text{TP}, \mathbb{V}, \mathbb{LV}, \mu_0, \mathbb{VP}, E)$ over a data type $(\mathbb{D}, \mathcal{DE}, \mathcal{DU})$, a data parameter valuation ζ and a timing parameter valuation γ , the semantics of $\gamma|\zeta(\mathcal{A})$ is given by the timed transition system (TTS) (S, s_0, \rightarrow) , with

- $S = L \times \mathbb{D}^M \times \mathbb{R}_{\geq 0}^H$, $s_0 = (\ell_0, \mu_0, \mathbf{0})$,
- \rightarrow consists of the discrete and (continuous) delay transition relations:

1. discrete transitions: $(\ell, \mu, \nu) \xrightarrow{e, \eta} (\ell', \mu', \nu')$, there exist $e = (\ell, tg, dg, a, R, \text{PDU}, \ell') \in E$ and a local variable valuation η defined exactly for $\text{Dom}(a)$, such that $\nu \models \gamma(tg)$, $(\mu, \eta) \models \zeta(dg)$, $\nu' = [\nu]_R$, and $\mu' = [\mu]_{\eta(\zeta(\text{PDU}))}$.
2. delay transitions: $(\ell, \mu, \nu) \xrightarrow{d} (\ell, \mu, \nu + d)$, with $d \in \mathbb{R}_{\geq 0}$.

Moreover we write $((\ell, \mu, \nu), (e, \eta, d), (\ell', \mu', \nu')) \in \rightarrow$ for a combination of a delay and discrete transition if $\exists \nu'' : (\ell, \mu, \nu) \xrightarrow{d} (\ell, \mu, \nu'') \xrightarrow{e, \eta} (\ell', \mu', \nu')$.

Given a TDA $\gamma|\zeta(\mathcal{A})$ with concrete semantics (S, s_0, \rightarrow) , we refer to the states of S as the *concrete states* of $\gamma|\zeta(\mathcal{A})$. A *run* of $\gamma|\zeta(\mathcal{A})$ is an alternating sequence of concrete states of $\gamma|\zeta(\mathcal{A})$ and triples of edges, local variable valuations and delays, starting from the initial state s_0 of the form $(\ell_0, \mu_0, \nu_0), (e_0, \eta, d_0), (\ell_1, \mu_1, \nu_1), \dots$ with $i = 0, 1, \dots$, $e_i \in E$, $d_i \in \mathbb{R}_{\geq 0}$ and $((\ell_i, \mu_i, \nu_i), (e_i, \eta_i, d_i), (\ell_{i+1}, \mu_{i+1}, \nu_{i+1})) \in \rightarrow$. Given such a run, the associated *timed data word* is $(a_1, \tau_1, \eta_1), (a_2, \tau_2, \eta_2), \dots$, where a_i is the action of edge e_{i-1} , η_i is the local variable valuation associated with that transition, and $\tau_i = \sum_{0 \leq j \leq i-1} d_j$, for $i = 1, 2, \dots$. For a timed data word w and a concrete state (ℓ, μ, ν) of $\gamma|\zeta(\mathcal{A})$, we write $(\ell_0, \mu_0, \mathbf{0}) \xrightarrow{w} (\ell, \mu, \nu)$ in $\gamma|\zeta(\mathcal{A})$ if w is associated with a run of $\gamma|\zeta(\mathcal{A})$ of the form $(\ell_0, \mu_0, \mathbf{0}), \dots, (\ell_n, \mu_n, \nu_n)$ with $(\ell_n, \mu_n, \nu_n) = (\ell, \mu, \nu)$. For a timed data word $w = (a_1, \tau_1, \eta_1), (a_2, \tau_2, \eta_2), \dots, (a_n, \tau_n, \eta_n)$, we denote $|w| = n$ and for any $i \in \{1, 2, \dots, n\}$, we denote $w(1, i) = (a_1, \tau_1, \eta_1), (a_2, \tau_2, \eta_2), \dots, (a_i, \tau_i, \eta_i)$.

A finite run is *accepting* if its last state (ℓ, μ, ν) is such that $\ell \in F$. The language $\mathcal{L}(\gamma|\zeta(\mathcal{A}))$ is defined to be the set of timed data words associated with all accepting runs of $\gamma|\zeta(\mathcal{A})$.

Example 5. Consider the PTDA in Fig. 2b over the data type for strings. Let $\gamma(\text{tp}) = 100$ and $\zeta(\text{vp}) = \text{Hakuchi.txt}$. An accepting run of the TDA $\gamma|\zeta(\mathcal{A})$ is: $(\ell_0, \emptyset, \nu_0), (e_0, \eta_0, 2046), (\ell_1, \emptyset, \nu_1), (e_1, \eta_1, 90), (\ell_1, \emptyset, \nu_2)(e_2, \eta_2, 30), (\ell_2, \emptyset, \nu_3)$, where \emptyset denotes a variable valuation over an empty domain (recall that $\mathbb{V} = \emptyset$ in Fig. 2b), $\nu_0(c) = 0$, $\nu_1(c) = 0$, $\nu_2(c) = 90$, $\nu_3(c) = 120$, e_0 is the upper edge from ℓ_0 to ℓ_1 , e_1 is the self-loop above ℓ_1 , e_2 is the lower edge from ℓ_1 to ℓ_2 , $\eta_0(f) = \eta_2(f) = \text{Hakuchi.txt}$, $\eta_1(f) = \text{Unagi.mp4}$, $\eta_0(m) = \eta_1(m) = \text{rw}$, and $\eta_2(m)$ is undefined (because $\text{Dom}(\text{close}) = \{f\}$).

The associated timed data word is $(\text{open}, 2046, \eta_0), (\text{open}, 2136, \eta_1), (\text{close}, 2166, \eta_2)$.

Since each action is associated with a set of local variables, given an ordering on this set, it is possible to see a given action and a variable valuation as a predicate: for example, assuming an ordering of \mathbb{LV} such as f precedes m , then open

with η_0 can be represented as `open(Hakuchi.txt, rw)`. Using this convention, the log in Fig. 2a corresponds exactly to this timed data word.

5 Symbolic Monitoring Against PTDA Specifications

In symbolic monitoring, in addition to the (observable) actions in Σ , we employ *unobservable* actions denoted by ε and satisfying $\text{Dom}(\varepsilon) = \emptyset$. We write Σ_ε for $\Sigma \sqcup \{\varepsilon\}$. We let η_ε be the local variable valuation such that $\eta_\varepsilon(lv)$ is undefined for any $lv \in \mathbb{LV}$. For a timed data word $w = (a_1, \tau_1, \eta_1), (a_2, \tau_2, \eta_2), \dots, (a_n, \tau_n, \eta_n)$ over Σ_ε , the projection $w \downarrow \Sigma$ is the timed data word over Σ obtained from w by removing any triple (a_i, τ_i, η_i) where $a_i = \varepsilon$. An edge $e = (\ell, tg, dg, a, R, \text{PDU}, \ell') \in E$ is *unobservable* if $a = \varepsilon$, and *observable* otherwise. The use of unobservable actions allows us to encode parametric timed pattern matching (see Sect. 5.3).

We make the following assumption on the PTDAs in symbolic monitoring.

Assumption 1. *The PTDA \mathcal{A} does not contain any loop of unobservable edges.*

5.1 Problem Definition

Roughly speaking, given a PTDA \mathcal{A} and a timed data word w , the symbolic monitoring problem asks for the set of pairs $(\gamma, \zeta) \in (\mathbb{Q}_+)^{\mathbb{TP}} \times \mathbb{D}^{\mathbb{VP}}$ satisfying $w(1, i) \in \gamma|\zeta(\mathcal{A})$, where $w(1, i)$ is a prefix of w . Since \mathcal{A} also contains unobservable edges, we consider w' which is w augmented by unobservable actions.

Symbolic monitoring problem:

INPUT: a PTDA \mathcal{A} over a data type $(\mathbb{D}, \mathcal{DE}, \mathcal{DU})$ and actions Σ_ε , and a timed data word w over Σ

PROBLEM: compute all the pairs (γ, ζ) of timing and data parameter valuations such that there is a timed data word w' over Σ_ε and $i \in \{1, 2, \dots, |w'|\}$ satisfying $w' \downarrow \Sigma = w$ and $w'(1, i) \in \mathcal{L}(\gamma|\zeta(\mathcal{A}))$. That is, it requires the validity domain $D(w, \mathcal{A}) = \{(\gamma, \zeta) \mid \exists w' : i \in \{1, 2, \dots, |w'|\}, w' \downarrow \Sigma = w \text{ and } w'(1, i) \in \mathcal{L}(\gamma|\zeta(\mathcal{A}))\}$.

Example 6. Consider the PTDA \mathcal{A} and the timed data word w shown in Fig. 1. The validity domain $D(w, \mathcal{A})$ is $D(w, \mathcal{A}) = D_1 \cup D_2$, where

$$D_1 = \{(\gamma, \zeta) \mid 0 \leq \gamma(\text{tp}) \leq 2, \zeta(\text{xp}) = c\} \text{ and } D_2 = \{(\gamma, \zeta) \mid 0 \leq \gamma(\text{tp}) \leq 1, \zeta(\text{xp}) = a\}.$$

For $w' = w(1, 3) \cdot (\varepsilon, \eta_\varepsilon, 2.9)$, we have $w' \in \mathcal{L}(\gamma|\zeta(\mathcal{A}))$ and $w' \downarrow \Sigma = w(1, 3)$, where γ and ζ are such that $\gamma(\text{tp}) = 1.8$ and $\zeta(\text{xp}) = c$, and $w(1, 3) \cdot (\varepsilon, \eta_\varepsilon, 2.9)$ denotes the juxtaposition.

For the data types in Example 2, the validity domain $D(w, \mathcal{A})$ can be represented by a constraint of finite size because the length $|w|$ of the timed data word is finite.

5.2 Online Algorithm

Our algorithm is *online* in the sense that it outputs $(\gamma, \zeta) \in D(w, \mathcal{A})$ as soon as its membership is witnessed, even before reading the whole timed data word w .

Let $w = (a_1, \tau_1, \eta_1), (a_2, \tau_2, \eta_2), \dots, (a_n, \tau_n, \eta_n)$ and \mathcal{A} be the timed data word and PTDA given in symbolic monitoring, respectively. Intuitively, after reading (a_i, τ_i, η_i) , our algorithm symbolically computes for all parameter valuations $(\gamma, \zeta) \in (\mathbb{Q}_+)^{\text{TP}} \times \mathbb{D}^{\text{VP}}$ the concrete states (ℓ, ν, μ) satisfying $(\ell_0, \mu_0, \mathbf{0}) \xrightarrow{w(1,i)} (\ell, \mu, \nu)$ in $\gamma|\zeta(\mathcal{A})$. Since \mathcal{A} has unobservable edges as well as observable edges, we have to add unobservable actions before or after observable actions in w . By Conf_i^o , we denote the configurations after reading (a_i, τ_i, η_i) and no unobservable actions are appended after (a_i, τ_i, η_i) . By Conf_i^u , we denote the configurations after reading (a_i, τ_i, η_i) and at least one unobservable action is appended after (a_i, τ_i, η_i) .

Definition 3 ($\text{Conf}_i^o, \text{Conf}_i^u$). *For a PTDA \mathcal{A} over actions Σ_ε , a timed data word w over Σ , and $i \in \{0, 1, \dots, |w|\}$ (resp. $i \in \{-1, 0, \dots, |w|\}$), Conf_i^o (resp. Conf_i^u) is the set of 5-tuples $(\ell, \nu, \gamma, \mu, \zeta)$ such that there is a timed data word w' over Σ_ε satisfying the following: (i) $(\ell_0, \mu_0, \mathbf{0}) \xrightarrow{w'} (\ell, \mu, \nu)$ in $\gamma|\zeta(\mathcal{A})$, (ii) $w' \downarrow \Sigma = w(1, i)$, (iii) The last action $a'_{|w'|}$ of w' is observable (resp. unobservable and its timestamp is less than τ_{i+1}).*

Algorithm 1. Outline of our algorithm for symbolic monitoring

Input: A PTDA $\mathcal{A} = (\Sigma_\varepsilon, L, \ell_0, F, \mathbb{C}, \text{TP}, \mathbb{V}, \text{LV}, \mu_0, \text{VP}, E)$ over a data type $(\mathbb{D}, \mathcal{DE}, \mathcal{DU})$ and actions Σ_ε , and a timed data word $w = (a_1, \tau_1, \eta_1), (a_2, \tau_2, \eta_2), \dots, (a_n, \tau_n, \eta_n)$ over Σ

Output: $\bigcup_{i \in \{1, 2, \dots, n+1\}} \text{Result}_i$ is the validity domain $D(w, \mathcal{A})$

```

1  $\text{Conf}_{-1}^u \leftarrow \emptyset; \text{Conf}_0^o \leftarrow \{(\ell_0, \mathbf{0}, \gamma, \mu_0, \zeta) \mid \gamma \in (\mathbb{Q}_+)^{\text{TP}}, \zeta \in \mathbb{D}^{\text{VP}}\}$ 
2 for  $i \leftarrow 1$  to  $n$  do
3   compute  $(\text{Conf}_{i-1}^u, \text{Conf}_i^o)$  from  $(\text{Conf}_{i-2}^u, \text{Conf}_{i-1}^o)$ 
4    $\text{Result}_i \leftarrow \{(\gamma, \zeta) \mid \exists (\ell, \nu, \gamma, \mu, \zeta) \in \text{Conf}_{i-1}^u \cup \text{Conf}_i^o. \ell \in F\}$ 
5   compute  $\text{Conf}_n^u$  from  $(\text{Conf}_{n-1}^u, \text{Conf}_n^o)$ 
6    $\text{Result}_{n+1} \leftarrow \{(\gamma, \zeta) \mid \exists (\ell, \nu, \gamma, \mu, \zeta) \in \text{Conf}_n^u. \ell \in F\}$ 

```

Algorithm 1 shows an outline of our algorithm for symbolic monitoring (see [35] for the full version). Our algorithm incrementally computes Conf_{i-1}^u and Conf_i^o (line 3). After reading (a_i, τ_i, η_i) , our algorithm stores the partial results $(\gamma, \zeta) \in D(w, \mathcal{A})$ witnessed from the accepting configurations in Conf_{i-1}^u and Conf_i^o (line 4). (We also need to try to take potential unobservable transitions and store the results from the accepting configurations *after* the last element of the timed data word (lines 5 and 6).)

Since $(\mathbb{Q}_+)^{\text{TP}} \times \mathbb{D}^{\text{VP}}$ is an infinite set, we cannot try each $(\gamma, \zeta) \in (\mathbb{Q}_+)^{\text{TP}} \times \mathbb{D}^{\text{VP}}$ and we use a symbolic representation for parameter valuations. Similarly to the

reachability synthesis of parametric timed automata [28], a set of clock and timing parameter valuations can be represented by a convex polyhedron. For variable valuations and data parameter valuations, we need an appropriate representation depending on the data type ($\mathbb{D}, \mathcal{DE}, \mathcal{DU}$). Moreover, for the termination of Algorithm 1, some operations on the symbolic representation are required.

Theorem 1 (termination). *For any PTDA \mathcal{A} over a data type $(\mathbb{D}, \mathcal{DE}, \mathcal{DU})$ and actions Σ_ε , and for any timed data word w over Σ , Algorithm 1 terminates if the following operations on the symbolic representation V_d of a set of variable and data parameter valuations terminate.*

1. restriction and update $\{([\mu]_{\eta(\zeta(\text{PDU}))}, \zeta) \mid \exists (\mu, \zeta) \in V_d. (\mu, \eta) \models \zeta(dg)\}$, where η is a local variable valuation, PDU is a parametric data update function, and dg is a data guard;
2. emptiness checking of V_d ;
3. projection $V_d \downarrow \mathbb{VP}$ of V_d to the data parameters \mathbb{VP} . \square

Example 7. For the data type for rationals in Example 2, variable and data parameter valuations V_d can be represented by convex polyhedra and the above operations terminate. For the data type for strings \mathbb{S} in Example 2, variable and data parameter valuations V_d can be represented by $\mathbb{S}^{|\mathbb{V}|} \times (\mathbb{S} \cup \mathcal{P}_{\text{fin}}(\mathbb{S}))^{|\mathbb{VP}|}$ and the above operations terminate, where $\mathcal{P}_{\text{fin}}(\mathbb{S})$ is the set of finite sets of \mathbb{S} .

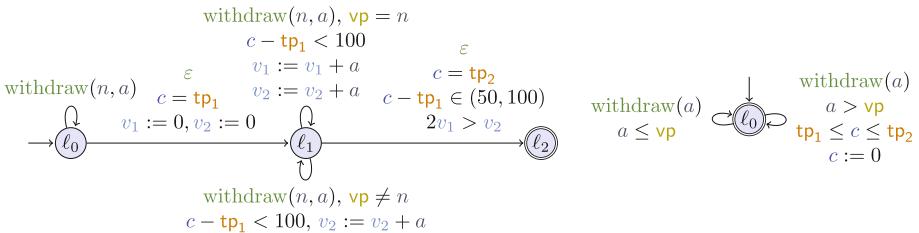


Fig. 3. PTDAs in DOMINANT (left) and PERIODIC (right)

5.3 Encoding Parametric Timed Pattern Matching

The symbolic monitoring problem is a generalization of the parametric timed pattern matching problem of [4]. Recall that parametric timed pattern matching aims at synthesizing timing parameter valuations and *start and end times in the log* for which a log segment satisfies or violates a specification. In our approach, by adding a clock measuring the absolute time, and two timing parameters encoding respectively the start and end date of the segment, one can easily infer the log segments for which the property is satisfied.

Consider the DOMINANT PTDA (left of Fig. 3). It is inspired by a monitoring of withdrawals from bank accounts of various users [15]. This PTDA monitors situations when a user withdraws more than half of the total withdrawals within a time window of $(50, 100)$. The actions are $\Sigma = \{\text{withdraw}\}$

and $\text{Dom}(\text{withdraw}) = \{n, a\}$, where n has a string value and a has an integer value. The string n represents a user name and the integer a represents the amount of the withdrawal by the user n . Observe that clock c is never reset, and therefore measures absolute time. The automaton can non-deterministically remain in ℓ_0 , or start to measure a log by taking the ε -transition to ℓ_1 checking $c = \text{tp}_1$, and therefore “remembering” the start time using timing parameter tp_1 . Then, whenever a user vp has withdrawn more than half of the accumulated withdrawals (data guard $2v_1 > v_2$) in a $(50, 100)$ time window (timed guard $c - \text{tp}_1 \in (50, 100)$), the automaton takes a ε -transition to the accepting location, checking $c = \text{tp}_2$, and therefore remembering the end time using timing parameter tp_2 .

6 Experiments

We implemented our symbolic monitoring algorithm in a tool SYMON in C++, where the domain for data is the strings and the integers. Our tool SYMON is distributed at <https://github.com/MasWag/symon>. We use PPL [8] for the symbolic representation of the valuations. We note that we employ an optimization to merge adjacent polyhedra in the configurations if possible. We evaluated our monitor algorithm against three original benchmarks: COPY in Fig. 1c; and DOMINANT and PERIODIC in Fig. 3. We conducted experiments on an Amazon EC2 c4.large instance (2.9 GHz Intel Xeon E5-2666 v3, 2 vCPUs, and 3.75 GiB RAM) that runs Ubuntu 18.04 LTS (64 bit).

6.1 Benchmark 1: Copy

Our first benchmark COPY is a monitoring of variable updates much like the scenario in [18]. The actions are $\Sigma = \{\text{update}\}$ and $\text{Dom}(\text{update}) = \{n, v\}$, where n has a string value representing the name of the updated variables and v has an integer value representing the updated value. Our set consists of 10 timed data words of length 4,000 to 40,000.

The PTDA in COPY is shown in Fig. 1c, where we give an additional constraint $3 < \text{tp} < 10$ on tp . The property encoded in Fig. 1c is “for any variable px , whenever an update of that variable occurs, then within tp time units, the value of b must be equal to that update”.

The experiment result is in Fig. 4. We observe that the execution time is linear to the number of the events and the memory usage is more or less constant with respect to the number of events.

6.2 Benchmark 2: Dominant

Our second benchmark is DOMINANT (Fig. 3 left). Our set consists of 10 timed data words of length 2,000 to 20,000. The experiment result is in Fig. 5. We observe that the execution time is linear to the number of the events and the memory usage is more or less constant with respect to the number of events.

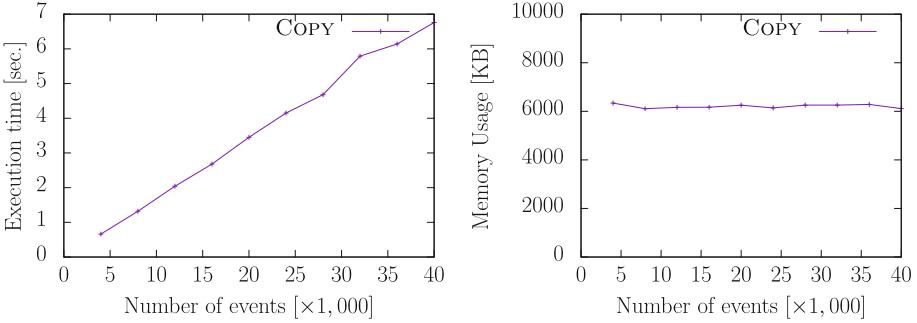


Fig. 4. Execution time (left) and memory usage (right) of COPY

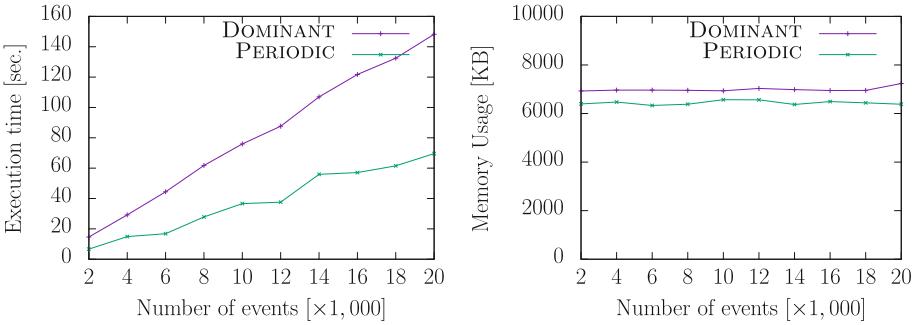


Fig. 5. Execution time (left) and memory usage (right) of DOMINANT and PERIODIC

6.3 Benchmark 3: Periodic

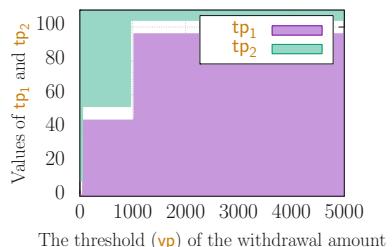
Our third benchmark PERIODIC is inspired by a parameter identification of periodic withdrawals from one bank account. The actions are $\Sigma = \{\text{withdraw}\}$ and $\text{Dom}(\text{withdraw}) = \{a\}$, where a has an integer value representing the amount of the withdrawal. We randomly generated a set consisting of 10 timed data words of length 2,000 to 20,000. Each timed data word consists of the following three kinds of periodic withdrawals:

shortperiod One withdrawal occurs every 5 ± 1 time units. The amount of the withdrawal is 50 ± 3 .

middleperiod One withdrawal occurs every 50 ± 3 time units. The amount of the withdrawal is 1000 ± 40 .

longperiod One withdrawal occurs every 100 ± 5 time units. The amount of the withdrawal is 5000 ± 20 .

The PTDA in PERIODIC is shown in the right of Fig. 3. The PTDA matches situations where, for any two successive withdrawals of amount more than vp , the duration between them is within $[tp_1, tp_2]$. By the symbolic monitoring, one can identify the period of the



periodic withdrawals of amount greater than vp is in $[\text{tp}_1, \text{tp}_2]$. An example of the validity domain is shown in the right figure.

The experiment result is in Fig. 5. We observe that the execution time is linear to the number of the events and the memory usage is more or less constant with respect to the number of events.

6.4 Discussion

First, a positive result is that our algorithm effectively performs symbolic monitoring on more than 10,000 actions in one or two minutes even though the PTDA feature both timing and data parameters. The execution time in COPY is 50–100 times smaller than that in DOMINANT and PERIODIC. This is because the constraint $3 < \text{tp} < 10$ in COPY is strict and the size of the configurations (i.e., Conf_i^o and Conf_i^u in Algorithm 1) is small. Another positive result is that in all of the benchmarks, the execution time is linear and the memory usage is more or less constant in the size of the input word. This is because the size of configurations (i.e., Conf_i^o and Conf_i^u in Algorithm 1) is bounded due to the following reason. In DOMINANT, the loop in ℓ_1 of the PTDA is deterministic, and because of the guard $c - \text{tp}_1 \in (50, 100)$ in the edge from ℓ_1 to ℓ_2 , the number of the loop edges at ℓ_1 in an accepting run is bounded (if the duration between two continuing actions are bounded as in the current setting). Therefore, $|\text{Conf}_i^o|$ and $|\text{Conf}_i^u|$ in Algorithm 1 are bounded. The reason is similar in COPY, too. In PERIODIC, since the PTDA is deterministic and the valuations of the amount of the withdrawals are in finite number, $|\text{Conf}_i^o|$ and $|\text{Conf}_i^u|$ in Algorithm 1 are bounded.

It is clear that we can design ad-hoc automata for which the execution time of symbolic monitoring can grow much faster (e.g., exponential in the size of input word). However, experiments showed that our algorithm monitors various interesting properties in a reasonable time.

COPY and DOMINANT use data and timing parameters as well as memory and aggregation; from Table 1, no other monitoring tool can compute the valuations satisfying the specification. We however used the parametric timed model checker IMITATOR [3] to try to perform such a synthesis, by encoding the input log as a separate automaton; but IMITATOR ran out of memory (on a 3.75 GiB RAM computer) for DOMINANT with $|w| = 2000$, while SYMON terminates in 14 s with only 6.9 MiB for the same benchmark. Concerning PERIODIC, the only existing work that can possibly accommodate this specification is [7]. While the precise performance comparison is interesting future work (their implementation is not publicly available), we do not expect our implementation be vastly outperformed: in [7], their tool times out (after 10 min) for a simple specification (“ $\mathbf{E}_{[0,s_2]} \mathbf{G}_{[0,s_1]}(x < p)$ ”) and a signal discretized by only 128 points.

For those problem instances which MONPOLY and DEJAVU can accommodate (which are simpler and less parametrized than our benchmarks), they tend to run much faster than ours. For example, in [26], it is reported that they can process a trace of length 1,100,004 in 30.3 s. The trade-off here is expressivity: for

example, DEJAVU does not seem to accommodate DOMINANT, because DEJAVU does not allow for aggregation. We also note that, while SYMON can be slower than MONPOLY and DEJAVU, it is fast enough for many scenarios of real-world online monitoring.

7 Conclusion and Perspectives

We proposed a symbolic framework for monitoring using parameters both in data and time. Logs can use timestamps and infinite domain data, while our monitor automata can use timing and variable parameters (in addition to clocks and local variables). In addition, our online algorithm can answer symbolically, by outputting all valuations (and possibly log segments) for which the specification is satisfied or violated. We implemented our approach into a prototype SYMON and experiments showed that our tool can effectively monitor logs of dozens of thousands of events in a short time.

Perspectives. Combining the BDDs used in [26] with some of our data types (typically strings) could improve our approach by making it even more symbolic. Also, taking advantage of the polarity of some parameters (typically the timing parameters, in the line of [17]) could improve further the efficiency.

We considered *infinite* domains, but the case of *finite* domains raises interesting questions concerning result representation: if the answer to a property is “neither **a** nor **b**”, knowing the domain is {**a**, **b**, **c**}, then the answer should be **c**.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. *Theor. Comput. Sci.* **126**(2), 183–235 (1994). [https://doi.org/10.1016/0304-3975\(94\)90010-8](https://doi.org/10.1016/0304-3975(94)90010-8)
2. Alur, R., Henzinger, T.A., Vardi, M.Y.: Parametric real-time reasoning. In: Kosaraju, S.R., Johnson, D.S., Aggarwal, A. (eds.) STOC, pp. 592–601. ACM, New York (1993). <https://doi.org/10.1145/167088.167242>
3. André, É., Fribourg, L., Kühne, U., Soulat, R.: IMITATOR 2.5: a tool for analyzing robustness in scheduling problems. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 33–36. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32759-9_6
4. André, É., Hasuo, I., Waga, M.: Offline timed pattern matching under uncertainty. In: Lin, A.W., Sun, J. (eds.) ICECCS, pp. 10–20. IEEE CPS (2018). <https://doi.org/10.1109/ICECCS2018.2018.00010>
5. Annpureddy, Y., Liu, C., Fainekos, G., Sankaranarayanan, S.: S-TALiRO: a tool for temporal logic falsification for hybrid systems. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 254–257. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19835-9_21
6. Asarin, E., Basset, N., Degorre, A.: Distance on timed words and applications. In: Jansen, D.N., Prabhakar, P. (eds.) FORMATS 2018. LNCS, vol. 11022, pp. 199–214. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-00151-3_12

7. Asarin, E., Donzé, A., Maler, O., Nickovic, D.: Parametric identification of temporal properties. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 147–160. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29860-8_12
8. 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). <https://doi.org/10.1016/j.scico.2007.08.001>
9. Bakhrirkin, A., Ferrère, T., Maler, O.: Efficient parametric identification for STL. In: HSCC, pp. 177–186. ACM (2018). <https://doi.org/10.1145/3178126.3178132>
10. Bakhrirkin, A., Ferrère, T., Maler, O., Ulus, D.: On the quantitative semantics of regular expressions over real-valued signals. In: Abate, A., Geeraerts, G. (eds.) FORMATS 2017. LNCS, vol. 10419, pp. 189–206. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-65765-3_11
11. Bakhrirkin, A., Ferrère, T., Nickovic, D., Maler, O., Asarin, E.: Online timed pattern matching using automata. In: Jansen, D.N., Prabhakar, P. (eds.) FORMATS 2018. LNCS, vol. 11022, pp. 215–232. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-00151-3_13
12. Barringer, H., Falcone, Y., Havelund, K., Reger, G., Rydeheard, D.: Quantified event automata: towards expressive and efficient runtime monitors. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 68–84. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32759-9_9
13. Basin, D.A., Klaedtke, F., Marinovic, S., Zalinescu, E.: Monitoring of temporal first-order properties with aggregations. *Form. Methods Syst. Des.* **46**(3), 262–285 (2015). <https://doi.org/10.1007/s10703-015-0222-7>
14. Basin, D.A., Klaedtke, F., Müller, S., Zalinescu, E.: Monitoring metric first-order temporal properties. *J. ACM* **62**(2), 15:1–15:45 (2015). <https://doi.org/10.1145/2699444>
15. Basin, D.A., Klaedtke, F., Zalinescu, E.: The MonPoly monitoring tool. In: Reger, G., Havelund, K. (eds.) RV-CuBES. Kalpa Publications in Computing, vol. 3, pp. 19–28. EasyChair (2017)
16. Bouajjani, A., Echahed, R., Robbana, R.: On the automatic verification of systems with continuous variables and unbounded discrete data structures. In: Antsaklis, P., Kohn, W., Nerode, A., Sastry, S. (eds.) HS 1994. LNCS, vol. 999, pp. 64–85. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-60472-3_4
17. Bozzelli, L., La Torre, S.: Decision problems for lower/upper bound parametric timed automata. *Form. Methods Syst. Des.* **35**(2), 121–151 (2009). <https://doi.org/10.1007/s10703-009-0074-0>
18. Brim, L., Dluhos, P., Safránek, D., Vejpustek, T.: STL*: extending signal temporal logic with signal-value freezing operator. *Inf. Comput.* **236**, 52–67 (2014). <https://doi.org/10.1016/j.ic.2014.01.012>
19. Dang, Z.: Pushdown timed automata: a binary reachability characterization and safety verification. *Theor. Comput. Sci.* **302**(1–3), 93–121 (2003). [https://doi.org/10.1016/S0304-3975\(02\)00743-0](https://doi.org/10.1016/S0304-3975(02)00743-0)
20. Deshmukh, J.V., Majumdar, R., Prabhu, V.S.: Quantifying conformance using the Skorokhod metric. *Form. Methods Syst. Des.* **50**(2–3), 168–206 (2017). <https://doi.org/10.1007/s10703-016-0261-8>
21. Donzé, A.: Breach, a toolbox for verification and parameter synthesis of hybrid systems. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 167–170. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_17

22. Donzé, A., Ferrère, T., Maler, O.: Efficient robust monitoring for STL. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 264–279. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_19
23. Donzé, A., Maler, O.: Robust satisfaction of temporal logic over real-valued signals. In: Chatterjee, K., Henzinger, T.A. (eds.) FORMATS 2010. LNCS, vol. 6246, pp. 92–106. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15297-9_9
24. Fages, F., Rizk, A.: On temporal logic constraint solving for analyzing numerical data time series. *Theor. Comput. Sci.* **408**(1), 55–65 (2008). <https://doi.org/10.1016/j.tcs.2008.07.004>
25. Fainekos, G.E., Pappas, G.J.: Robustness of temporal logic specifications for continuous-time signals. *Theor. Comput. Sci.* **410**(42), 4262–4291 (2009). <https://doi.org/10.1016/j.tcs.2009.06.021>
26. Havelund, K., Peled, D., Ulus, D.: First order temporal logic monitoring with BDDs. In: Stewart, D., Weissenbacher, G. (eds.) FMCAD, pp. 116–123. IEEE (2017). <https://doi.org/10.23919/FMCAD.2017.8102249>
27. Jakšić, S., Bartocci, E., Grosu, R., Nguyen, T., Ničković, D.: Quantitative monitoring of STL with edit distance. *Form. Methods Syst. Des.* **53**(1), 83–112 (2018). <https://doi.org/10.1007/s10703-018-0319-x>
28. Jovanović, A., Lime, D., Roux, O.H.: Integer parameter synthesis for real-time systems. *IEEE Trans. Softw. Eng.* **41**(5), 445–461 (2015). <https://doi.org/10.1109/TSE.2014.2357445>
29. Quaas, K.: Verification for timed automata extended with discrete data structure. *Log. Methods Comput. Sci.* **11**(3) (2015). [https://doi.org/10.2168/LMCS-11\(3:20\)2015](https://doi.org/10.2168/LMCS-11(3:20)2015)
30. Reger, G., Cruz, H.C., Rydeheard, D.: MARQ: monitoring at runtime with QEA. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 596–610. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_55
31. Ulus, D.: MONTRE: a tool for monitoring timed regular expressions. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017, Part I. LNCS, vol. 10426, pp. 329–335. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_16
32. Ulus, D., Ferrère, T., Asarin, E., Maler, O.: Timed pattern matching. In: Legay, A., Bozga, M. (eds.) FORMATS 2014. LNCS, vol. 8711, pp. 222–236. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10512-3_16
33. Ulus, D., Ferrère, T., Asarin, E., Maler, O.: Online timed pattern matching using derivatives. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 736–751. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_47
34. Ulus, D., Maler, O.: Specifying timed patterns using temporal logic. In: HSCC, pp. 167–176. ACM (2018). <https://doi.org/10.1145/3178126.3178129>
35. Waga, M., André, É., Hasuo, I.: Symbolic monitoring against specifications parametric in time and data. CoRR abs/1905.04486 (2019). [arxiv:1905.04486](https://arxiv.org/abs/1905.04486)
36. Waga, M., Hasuo, I., Suenaga, K.: Efficient online timed pattern matching by automata-based skipping. In: Abate, A., Geeraerts, G. (eds.) FORMATS 2017. LNCS, vol. 10419, pp. 224–243. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-65765-3_13

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





STAMINA: STochastic Approximate Model-Checker for INfinite-State Analysis

Thakur Neupane¹ , Chris J. Myers² , Curtis Madsen³ , Hao Zheng⁴ , and Zhen Zhang¹

¹ Utah State University, Logan, UT, USA
thakur.neupane@aggiemail.usu.edu
zhen.zhang@usu.edu

² University of Utah, Salt Lake City, UT, USA
myers@ece.utah.edu

³ Boston University, Boston, MA, USA
ckmadsen@bu.edu

⁴ University of South Florida, Tampa, FL, USA
haozheng@usf.edu



Abstract. Stochastic model checking is a technique for analyzing systems that possess probabilistic characteristics. However, its scalability is limited as probabilistic models of real-world applications typically have very large or infinite state space. This paper presents a new infinite state CTMC model checker, STAMINA, with improved scalability. It uses a novel state space approximation method to reduce large and possibly infinite state CTMC models to finite state representations that are amenable to existing stochastic model checkers. It is integrated with a new property-guided state expansion approach that improves the analysis accuracy. Demonstration of the tool on several benchmark examples shows promising results in terms of analysis efficiency and accuracy compared with a state-of-the-art CTMC model checker that deploys a similar approximation method.

Keywords: Stochastic model checking · Infinite-state · Markov chains

1 Introduction

Stochastic model checking is a formal method that designers and engineers can use to determine the likelihood of *safety* and *liveness* properties. Checking properties using numerical model checking techniques requires enumerating the state space of the system to determine the probability that the system is in any given state at a desired time [17]. Real-world applications often have very large or even infinite state spaces.

Numerous state representation, reduction, and approximation methods have been proposed. Symbolic model checking based on *multi-terminal binary decision diagrams* (MTBDDs) [23] has achieved success in representing large *Markov Decision Process* (MDP) models with a few distinct probabilistic choices at each state, e.g., the shared coin protocol [3]. MTBDDs, however, are often inefficient for models with many different and distinct probability/rate values due to the inefficient representation of solution

vectors. *Continuous-time Markov chain* (CTMC) models, whose state transition rate is a function of state variables, generally contain many distinct rate values. As a result, symbolic model checkers can run out of memory while verifying a typical CTMC model with as few as 73,000 states [23]. State reduction techniques, such as bisimulation minimization [7, 8, 14], abstraction [6, 12, 14, 20], symmetry reduction [5, 16], and partial order reduction [9] have been mainly extended to discrete-time, finite-state probabilistic systems. The three-valued abstraction [14] can reduce large, finite-state CTMCs. It may, however, provide inconclusive verification results due to abstraction.

To the best of our knowledge, only a few tools can analyze infinite-state probabilistic models, namely, STAR [19] and INFAMY [10]. The STAR tool primarily analyzes biochemical reaction networks. It approximates solutions to the *chemical master equation* (CME) using the *method of conditional moments* (MCM) [11] that combines moment-based and state-based representations of probability distributions. This hybrid approach represents species with low concentrations using a discrete stochastic description and numerically integrates a small master equation using the fourth order Runge-Kutta method over a small time interval [2]; and solves a system of conditional moment equations for higher concentration species, conditioned on the low concentration species. This method has been optimized to drop unlikely states and add likely states on-the-fly. STAR relies on a well-structured underlying Markov process with small sensitivity on the transient distribution. Also, it mainly reports state reachability probabilities, instead of checking a given probabilistic property. INFAMY is a truncation-based approach that explores the model's state space up to a certain finite depth k . The truncated state space still grows exponentially with respect to exploration depth. Starting from the initial state, breadth-first state search is performed up to a certain finite depth. The error probability computed during the model checking depends on the depth of state exploration. Therefore, higher exploration depth generally incurs lower error probability.

This paper presents a new infinite-state stochastic model checker, *STochastic Approximate Model-checker for INfinite-state Analysis* (STAMINA). Our tool also takes a truncation-based approach. In particular, it maintains a probability estimate of each path being explored in the state space, and when the currently explored path probability drops below a specified threshold, it halts exploration of this path. All transitions exiting this state are redirected to an absorbing state. After all paths have been explored or truncated, transient Markov chain analysis is applied to determine the probability of a transient property of interest specified using *Continuous Stochastic Logic* (CSL) [4]. The calculated probability forms a lower bound on the probability, while the upper bound also includes the probability of the absorbing state. The actual probability of the CSL property is guaranteed to be within this range. An initial version of our tool and preliminary results are reported in [22]. Since that paper, our tool has been tightly integrated within the PRISM model checker [18] to improve performance, and we have also developed a new property-guided state expansion technique to expand the state space to tighten the reported probability range incrementally. This paper reports our results, which show significant improvement on both efficiency and verification accuracy over several non-trivial case studies from various application domains.

2 STAMINA

Figure 1 presents the architecture of STAMINA. Based on a user-specified probability threshold \varkappa (kappa), it first constructs a finite-state CTMC model $\mathcal{C}_{\downarrow \varkappa}$ from the original infinite-state CTMC model \mathcal{C} using the state space approximation method presented in Sect. 2.1. $\mathcal{C}_{\downarrow \varkappa}$ is then checked using the PRISM explicit-state model checker against a given CSL property $P_{\sim p}(\phi)$, where $\sim \in \{<, >, \leqslant, \geqslant\}$ and $p \in [0, 1]$ (for cases where it is desired that a predicate be true within a certain probability bound) or $P_{=?}(\phi)$ (for cases where it is desired that the exact probability of the predicate being true be calculated). Lower- and upper-bound probabilities that ϕ holds, namely, P_{min} and P_{max} , are then obtained, and their difference, i.e., $(P_{max} - P_{min})$, is the probability accumulated in the absorbing state x_{abs} which abstracts all the states not included in the current state space. If $p \in [P_{min}, P_{max}]$, it is not known whether $P_{\sim p}(\phi)$ holds. If exact probability is of interest and the probability range is larger than the user-defined precision ϵ , i.e., $(P_{max} - P_{min}) > \epsilon$, then the method does not give a meaningful result.

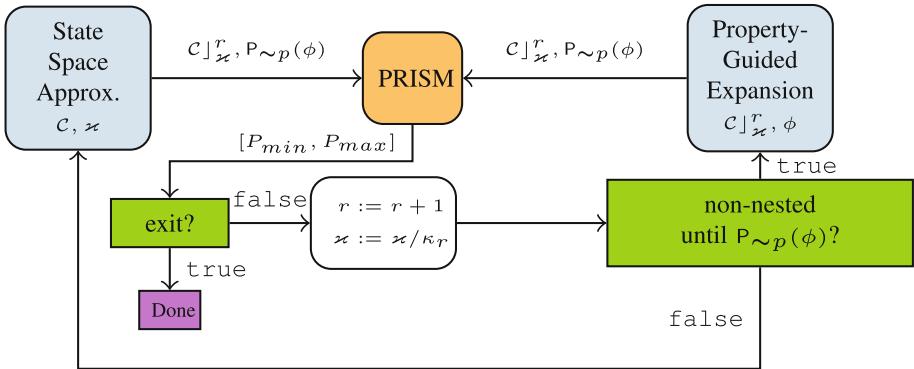


Fig. 1. Architecture of STAMINA.

For an inconclusive verification result from the previous step, STAMINA applies a property-guided approach, described in Sect. 2.2, to further expand $\mathcal{C}_{\downarrow \varkappa}$, provided $P_{\sim p}(\phi)$ is a non-nested “until” formula; otherwise, it uses the previous method to expand the state space. Note that \varkappa also drops by the reduction factor κ_r to enable states that were previously ignored due to a low probability estimate to be included in the current state expansion. The expanded CTMC model $\mathcal{C}_{\downarrow \varkappa}$ is then checked to obtain a new probability bound $[P_{min}, P_{max}]$. This iterative process repeats until one of the following conditions holds: (1) the target probability p falls outside the probability bound $[P_{min}, P_{max}]$, (2) the probability bound is sufficiently small, i.e., $(P_{max} - P_{min}) < \epsilon$, or (3) a maximal number of iterations N has been reached ($r \geq N$).

2.1 State Space Approximation

The state space approximation method [22] truncates the state space based on a user-specified reachability threshold \varkappa . During state exploration, the reachability-value func-

tion, $\hat{\kappa} : \mathbf{X} \rightarrow \mathbb{R}^+$, estimates the probability of reaching a state on-the-fly, and is compared against \varkappa to determine whether the state search should terminate. Only states with a higher reachability-value than the reachability threshold are explored further.

Figure 2 illustrates the standard *breadth first search* (BFS) state exploration for reachability threshold $\varkappa = 0.25$. It starts from the initial state whose reachability-value i.e., $\hat{\kappa}(x_0)$, is initialized to 1.0 as shown in Fig. 2a. In the first step, two new states x_1 and x_4 are generated and their reachability-values are 0.8 and 0.2, respectively, as shown in Fig. 2b. The reachability-value in x_0 is distributed to its successor states, based on the probability of outgoing transitions from x_0 to its successor state. For the next step, only state x_1 is scheduled for exploration because $\hat{\kappa}(x_1) \geq \varkappa$. Note that the transition from x_4 to x_0 is executed because x_0 is already in the explored set. Expanding x_1 leads to two new states, namely x_2 and x_5 as shown in Fig. 2c, from which only x_5 is scheduled for further exploration. This leads to the generation of x_6 and x_9 shown in Fig. 2d. State exploration terminates after Fig. 2e since both newly generated states have reachability-values less than 0.25. States x_2 , x_4 , x_6 and x_9 are marked as terminal states. During state exploration, the reachability-value update is performed every time a new incoming path is added to a state because a new incoming path can add its contribution to the state, potentially bringing the reachability-value above \varkappa , which in turn changes a terminal state to be non-terminal. When the truncated CTMC model $\mathcal{C}|_{\varkappa}$ is analyzed, it introduces some error in the probability value of the property under verification, because of leakage the probability (i.e., cumulative path probabilities of reaching states not included in the explored state space) during the CTMC analysis. To

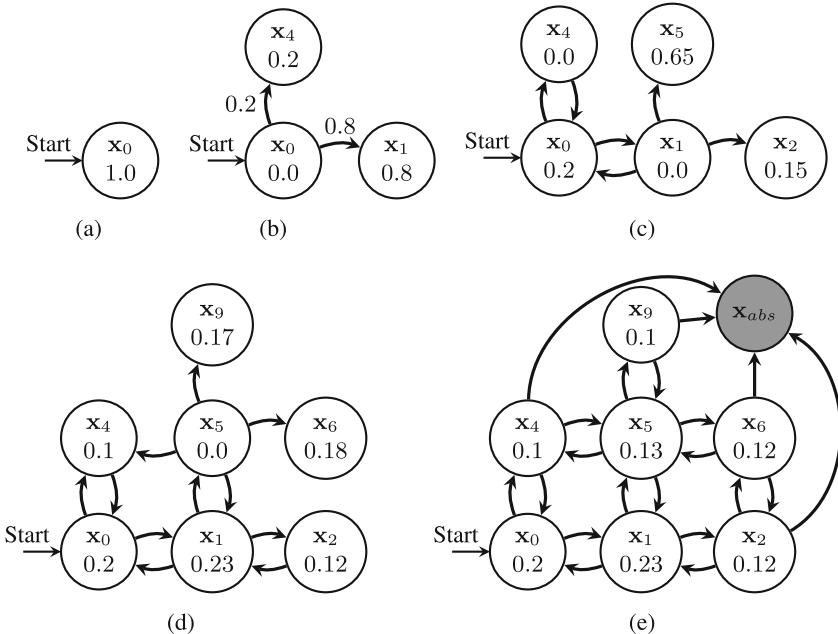


Fig. 2. State space approximation.

account for probability loss, an abstract absorbing state \mathbf{x}_{abs} is created as the sole successor state for all terminal states on each truncated path. Figure 2e shows the addition of the absorbing state.

2.2 Property Based State Space Exploration

This paper introduces a property-guided state expansion method, in order to efficiently obtain a tightened probability bound. Since all non-nested CSL path formulas ϕ (except those containing the “next” operator) derive from the “until” formula, $\Phi \mathcal{U}^I \Psi$, construction of the set of terminal states for further expansion boils down to eliminating states that are known to satisfy or dissatisfy $\Phi \mathcal{U} \Psi$. Given a state graph, a path starting from the initial state can never satisfy $\Phi \mathcal{U} \Psi$, if it includes a state satisfying $\neg\Phi \wedge \neg\Psi$. Also, if a path includes a state satisfying Ψ , satisfiability of $\Phi \mathcal{U} \Psi$ can be determined without further expanding this path beyond the first Ψ -state. Our property-guided state space expansion method identifies the path prefixes, from which satisfiability of $\Phi \mathcal{U} \Psi$ can be determined, and shortens them by making the last state of each prefix absorbing based on the satisfiability of $(\neg\Phi \vee \Psi)$. Only the non-absorbing states whose path probability is greater than the state probability estimate threshold \varkappa are expanded further. For detailed algorithms of STAMINA, readers are encouraged to read [21].

3 Results

This section presents results on the following case studies to illustrate the accuracy and efficiency of STAMINA: a genetic toggle switch [20,22]; the following examples from the PRISM benchmark suite [15]: grid world robot, cyclic server polling system, and tandem queuing network; and the Jackson queuing network from INFAMY case studies [1]. All case studies are evaluated on STAMINA and INFAMY, except the genetic toggle switch¹. Experiments are performed on a 3.2 GHz AMD Debian Linux PC with six cores and 64 GB of RAM. For all experiments, the maximal number of iterations N is set to 10, and the reduction factor κ_r is set to 1000. All experiments terminate due to $(P_{max} - P_{min}) < \epsilon$, where $\epsilon = 10^{-3}$, before they reach N . STAMINA is freely available at: <https://github.com/formal-verification-research/stamina>.

We compare the runtime, state size, and verification results between STAMINA and INFAMY using the same precision $\epsilon = 10^{-3}$. For all tables in this section, column \varkappa reports the probability estimate threshold used to terminate state generation in STAMINA. The state space size is listed in column $|\mathcal{G}|(K)$, where K indicates one thousand states. Column $T(C/A)$ reports the state space construction (C) and analysis (A) time in seconds. For STAMINA, the total construction and analysis time is the cumulation of runtime for all \varkappa values for a model configuration. Columns P_{min} and P_{max} list the lower and upper probability bounds for the property under verification, and column P lists the single probability value (within the precision ϵ) reported by INFAMY. We select the best runtime reported by three configurations of INFAMY. The improvement in state size (column $|\mathcal{G}|(X)$) and runtime (column $T(\%)$) are represented

¹ INFAMY generates arithmetic errors on the genetic toggle switch model.

by the ratio of state count generated by INFAMY to that of STAMINA (higher is better) and percentage improvement in runtime (higher is better), respectively.

Genetic Toggle Switch. The genetic toggle switch circuit model has two inputs, aTc and IPTG. It can be set to the OFF state by supplying it with aTc and can be set to the ON state by supplying it with IPTG [20]. Two important properties for a toggle switch circuit are the response time and the failure rate. The first experiments set IPTG to 100 to measure the toggle switch’s response time. It should be noted that the input value of 100 molecules of IPTG is chosen to ensure that the circuit switches to the ON state. The later experiments initialize IPTG to 0 to compute the failure rate, i.e., the probability that the circuit changes state erroneously within a cell cycle of 2,100 s (an approximation of the cell cycle in *E. coli* [24]). Initially, LacI is set to 60 and TetR is set to 0 for both experiments. The CSL property used for both experiments, $P_{=?} [\text{true } \mathcal{U}^{\leq 2100} (\text{TetR} > 40 \wedge \text{LacI} < 20)]$, describes the probability of the circuit switching to the ON state within a cell cycle of 2,100 s. The ON state is defined as LacI below 20 and TetR above 40 molecules.

Table 1. Verification results for genetic toggle switch.

IPTG	STAMINA					Remark
	\varkappa	$ \mathcal{G} $	$T(C/A)$	P_{min}	P_{max}	
100	10^{-3}	1,127	0.15/0.67	0.000000	0.999671	Property guided
	10^{-6}	4,461	0.43/2.84	0.966947	0.992908	
	10^{-9}	7,163	0.43/5.25	0.991738	0.991797	
100	10^{-6}	5,171	0.17/1.90	0.977942	0.992850	Property agnostic
	10^{-9}	8,908	0.18/3.74	0.991739	0.991797	
0	10^{-3}	182	0.05/0.07	0.000000	0.697500	Property guided
	10^{-6}	2,438	0.16/1.08	0.008814	0.060424	
	10^{-9}	4,284	0.09/2.12	0.013097	0.013609	
0	10^{-6}	2,446	0.16/1.05	0.009169	0.060420	Property agnostic
	10^{-9}	4,820	0.13/2.13	0.013097	0.013609	

The property-agnostic state space is generated with the probability estimate threshold $\varkappa = 10^{-3}$. Table 1 shows large probability bounds: $[0, 0.999671]$ for IPTG = 100 and $[0, 0.6975]$ for IPTG = 0. It is obvious that they are significantly inaccurate w.r.t. the precision ϵ of 10^{-3} . The \varkappa is then reduced to 10^{-6} and state generation switches to the property-guided state expansion mode, where the CSL property is used to guide state exploration, based on the previous state graph. Each state expansion step reduces the \varkappa value by a factor of $\kappa_r = 1000$. To measure the effectiveness of the property-guided state expansion approach, we compare state graphs generated with and without the property-guided state expansion, as indicated by the “property agnostic” and “property guided” rows in the table. Property-guided state expansion reduces the size of the state space without losing the analysis precision for the same value of \varkappa . Specifically,

the state expansion approach reduces the state space by almost 20% for the response rate experiment.

Robot World. This case study considers a robot moving in an n -by- n grid and a janitor moving in a larger grid Kn -by- Kn , where the constant K is used to significantly scale up the state space. The robot starts from the bottom left corner to reach the top right corner. The janitor moves around randomly. Either the robot or janitor can occupy one grid location at any given time. The robot also randomly communicates with the base station. The property of interest is the probability that the robot reaches the top right corner within 100 time units while periodically communicating with the base station, encoded as $P=? [(P_{\geq 0.5} [\text{true } U^{\leq 7} \text{ communicate}]) U^{\leq 100} \text{ goal }]$.

Table 2 provides a comparison of results for $K = 1024, 64$ and $n = 64, 32$. For smaller grid size i.e, 32-by-32, the robot can reach the goal with a high probability of 97.56%. Where as for a larger value of $n = 64$ and $K = 64$, the robot is not able to reach the goal with considerable probability. STAMINA generates precise results that are similar to INFAMY, while exploring less than half of states with shorter runtime.

Table 2. Comparison between STAMINA and INFAMY.

Model	Params	STAMINA				INFAMY				Improvement	
		$ \mathcal{G} (K)$	$T(C/A)$	P_{min}	P_{max}	$ \mathcal{G} (K)$	$T(C/A)$	P	$ \mathcal{G} (X)T(\%)$		
Robot (n/K)	32/64	696	41/279	0.975	0.975	1, 591	492/18	0.975	2.3	37.3	
	32/1024	696	41/258	0.975	0.975	1, 591	501/18	0.975	2.3	42.4	
	64/64	2, 273	135/669	$1.46e-4$	$1.68e-4$	5, 088	1, 625/53	$1.5e-4$	2.2	52.1	
	64/1024	2, 273	132/621	$1.46e-4$	$1.68e-4$	5, 088	1, 625/53	$1.5e-4$	2.2	55.2	
Jackson (N/λ)	4/5	201	22/51	0.865	0.865	635	109/5	0.865	3.2	36.1	
	5/5	2, 539	990/996	0.819	0.819	7, 029	1668/108	0.819	2.8	-11.8	
Polling (N)	12	19	3/21	1.0	1.0	74	1/2	1.0	3.9	-732.2	
	16	57	18/70	1.0	1.0	1, 573	5/54	1.0	27.6	-48.2	
	20	113	30/77	1.0	1.0	31, 457	151/1347	1.0	278.4	92.9	
Tandem (c)	2047	33	1/41	0.498	0.498	2, 392	3/38	0.498	72.5	-1.4	
	4095	66	1/141	0.499	0.499	9, 216	11/265	0.499	139.6	48.7	

Jackson Queuing Network. A Jackson queuing network consists of N interconnected nodes (queues) with infinite queue capacity. Initially, all queues are considered empty. Each station is connected to a single server which distributes the arrived jobs to different stations. Customers arrive as a Poisson stream with intensity λ for N queues. The model is taken from [10, 13]. We compute the probability that, within 10 time units, the first queue has more than 3 jobs and the second queue has more than 5 jobs, given by $P=? [\text{true } U^{\leq 10} (jobs_1 \geq 4 \wedge jobs_2 \geq 6)]$.

Table 2 summarizes the results for this model. STAMINA uses roughly equal time to construct and analyze the model for $N = 5$, whereas INFAMY takes significantly longer to construct the state space, making it slower in overall runtime. For $N = 4$, STAMINA is faster in generating verification results In both configurations, STAMINA only explores approximately one third of the states explored by INFAMY.

Cyclic Server Polling System. This case study is based on a cyclic server attending N stations. We consider the probability that station one is polled within 10 time units, $P_{=?} [\text{true } U^{\leq 10} \text{ station1_polled}]$. Table 2 summarizes the verification results for $N = 12, 16, 20$. The probability of station one being polled within 10 s is 1.0 for all configurations. Similar to previous case studies, STAMINA explores significantly smaller state space. The advantage of STAMINA in terms of runtime starts to manifest as the size of model (and hence the state space size) grows.

Tandem Queuing Network. A tandem queuing network is the simplest interconnected queuing network of two finite capacity (c) queues with one server each [18]. Customers join the first queue and enter the second queue immediately after completing the service. This paper considers the probability that the first queue becomes full in 0.25 time units, depicted by the CSL property $P_{=?} [\text{true } U^{\leq 0.25} \text{ queue1_full}]$.

As seen in Table 2, there is almost fifty percent probability that the first queue is full in 0.25 s irrespective of the queue capacity. As in the polling server, STAMINA explores significantly smaller state space. The runtime is similar for model with smaller queue capacity ($c = 2047$). But the runtime improves as the queue capacity is increased.

4 Conclusions

This paper presents an infinite-state stochastic model checker, STAMINA, that uses path probability estimates to generate states with high probability and truncate unlikely states based on a specified threshold. Initial state construction is property agnostic, and the state space is used for stochastic model checking of a given CSL property. The calculated probability forms a lower and upper bound on the probability for the CSL property, which is guaranteed to include the actual probability. Next, if finer precision of the probability bound is required, it uses a property-guided state expansion technique to explore states to tighten the reported probability range incrementally. Implementation of STAMINA is built on top of the PRISM model checker with tight integration to its API. Performance and accuracy evaluation is performed on case studies taken from various application domains, and shows significant improvement over the state-of-art infinite-state stochastic model checker INFAMY. For future work, we plan to investigate methods to determine the reduction factor on-the-fly based on the probability bound. Another direction is to investigate heuristics to further improve the property-guided state expansion, as well as, techniques to dynamically remove unlikely states.

Acknowledgment. Chris Myers is supported by the National Science Foundation under CCF-1748200. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

References

1. <https://depend.cs.uni-saarland.de/tools/infamy/casestudies/>
2. Andreychenko, A., Mikeev, L., Spieler, D., Wolf, V.: Parameter identification for Markov models of biochemical reactions. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 83–98. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_8

3. Aspnes, J., Herlihy, M.: Fast randomized consensus using shared memory. *J. Algorithms* **11**(3), 441–461 (1990)
4. Aziz, A., Sanwal, K., Singhal, V., Brayton, R.: Model-checking continuous-time Markov chains. *ACM Trans. Comput. Logic* **1**(1), 162–170 (2000)
5. Donaldson, A.F., Miller, A.: Symmetry reduction for probabilistic model checking using generic representatives. In: Graf, S., Zhang, W. (eds.) ATVA 2006. LNCS, vol. 4218, pp. 9–23. Springer, Heidelberg (2006). https://doi.org/10.1007/11901914_4
6. Ficher, H., Leucker, M., Wolf, V.: *Don't know* in probabilistic systems. In: Valmari, A. (ed.) SPIN 2006. LNCS, vol. 3925, pp. 71–88. Springer, Heidelberg (2006). https://doi.org/10.1007/11691617_5
7. Fisler, K., Vardi, M.Y.: Bisimulation and model checking. In: Pierre, L., Kropf, T. (eds.) CHARME 1999. LNCS, vol. 1703, pp. 338–342. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48153-2_29
8. Fisler, K., Vardi, M.Y.: Bisimulation minimization and symbolic model checking. *Form. Methods Syst. Des.* **21**(1), 39–78 (2002)
9. Groesser, M., Baier, C.: Partial order reduction for Markov decision processes: a survey. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 408–427. Springer, Heidelberg (2006). https://doi.org/10.1007/11804192_19
10. Hahn, E.M., Hermanns, H., Wachter, B., Zhang, L.: INFAMY: an infinite-state Markov model checker. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 641–647. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_49
11. Hasenauer, J., Wolf, V., Kazeroonian, A., Theis, F.J.: Method of conditional moments (MCM) for the chemical master equation. *J. Math. Biol.* **69**(3), 687–735 (2014). <https://doi.org/10.1007/s00285-013-0711-5>
12. Hermanns, H., Wachter, B., Zhang, L.: Probabilistic CEGAR. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 162–175. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70545-1_16
13. Jackson, J.: Networks of waiting lines. *Oper. Res.* **5**, 518–521 (1957)
14. Katoen, J.-P., Klink, D., Leucker, M., Wolf, V.: Three-valued abstraction for continuous-time Markov chains. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 311–324. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73368-3_37
15. Kwiatkowska, M., Norman, G., Parker, D.: The prism benchmark suite. In: International Conference on (QEST) Quantitative Evaluation of Systems, vol. 00, pp. 203–204, September 2012. <https://doi.org/10.1109/QEST.2012.14>, doi.ieeecomputersociety.org/10.1109/QEST.2012.14
16. Kwiatkowska, M., Norman, G., Parker, D.: Symmetry reduction for probabilistic model checking. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 234–248. Springer, Heidelberg (2006). https://doi.org/10.1007/11817963_23
17. Kwiatkowska, M., Norman, G., Parker, D.: Stochastic model checking. In: Bernardo, M., Hillston, J. (eds.) SFM 2007. LNCS, vol. 4486, pp. 220–270. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-72522-0_6
18. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_47
19. Lapin, M., Mikeev, L., Wolf, V.: Shave: Stochastic hybrid analysis of Markov population models. In: Proceedings of the 14th International Conference on Hybrid Systems: Computation and Control, HSCC 2011, pp. 311–312. ACM, New York (2011)
20. Madsen, C., Zhang, Z., Roehner, N., Winstead, C., Myers, C.: Stochastic model checking of genetic circuits. *J. Emerg. Technol. Comput. Syst.* **11**(3), 23:1–23:21 (2014). <https://doi.org/10.1145/2644817>, http://doi.acm.org/10.1145/2644817

21. Neupane, T.: STAMINA: STochastic approximate model-checker for INfinite-state analysis. Master's thesis, Utah State University, May 2019
22. Neupane, T., Zhang, Z., Madsen, C., Zheng, H., Myers, C.J.: Approximation techniques for stochastic analysis of biological systems. In: Liò, P., Zuliani, P. (eds.) Automated Reasoning for Systems Biology and Medicine. CB, vol. 30, pp. 327–348. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17297-8_12
23. Parker, D.: Implementation of symbolic model checking for probabilistic systems. Ph.D. thesis, University of Birmingham (2002)
24. Zheng, H., et al.: Interrogating the escherichia coli cell cycle by cell dimension perturbations. Proc. Natl. Acad. Sci. **113**(52), 15000–15005 (2016)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.



Dynamical, Hybrid, and Reactive Systems



Local and Compositional Reasoning for Optimized Reactive Systems

Mitesh Jain^{1(✉)} and Panagiotis Manolios²

¹ Synopsys Inc., Mountain View, USA

mitesh.jain@synopsys.com

² Northeastern University, Boston, USA

pete@ccs.neu.edu

Abstract. We develop a compositional, algebraic theory of skipping refinement, as well as local proof methods to effectively analyze the correctness of optimized reactive systems. A verification methodology based on refinement involves showing that any infinite behavior of an optimized low-level implementation is a behavior of the high-level abstract specification. Skipping refinement is a recently introduced notion to reason about the correctness of optimized implementations that run faster than their specifications, *i.e.*, a step in the implementation can skip multiple steps of the specification. For the class of systems that exhibit bounded skipping, existing proof methods have been shown to be amenable to mechanized verification using theorem provers and model-checkers. However, reasoning about the correctness of reactive systems that exhibit unbounded skipping using these proof methods requires reachability analysis, significantly increasing the verification effort. In this paper, we develop two new sound and complete proof methods for skipping refinement. Even in presence of unbounded skipping, these proof methods require only local reasoning and, therefore, are amenable to mechanized verification. We also show that skipping refinement is compositional, so it can be used in a stepwise refinement methodology. Finally, we illustrate the utility of the theory of skipping refinement by proving the correctness of an optimized event processing system.

1 Introduction

Reasoning about the correctness of a reactive system using refinement involves showing that any (infinite) observable behavior of a low-level, optimized implementation is a behavior allowed by the simple, high-level abstract specification. Several notions of refinement like trace containment, (bi)simulation refinement, stuttering (bi)simulation refinement, and skipping refinement [4, 10, 14, 20, 22] have been proposed in the literature to directly account for the difference in the abstraction levels between a specification and an implementation. Two attributes of crucial importance that enable us to effectively verify complex reactive systems using refinement are: (1) Compositionality: this allows us to decompose a monolithic proof establishing that a low-level concrete implementation refines

© The Author(s) 2019

I. Dillig and S. Tasiran (Eds.): CAV 2019, LNCS 11561, pp. 553–571, 2019.

https://doi.org/10.1007/978-3-030-25540-4_32

a high-level abstract specification into a sequence of simpler refinement proofs, where each of the intermediate refinement proof can be performed independently using verification tools best suited for it; (2) Effective proof methods: analyzing the correctness of a reactive system requires global reasoning about its infinite behaviors, a task that is often difficult for verification tools. Hence it is crucial that the refinement-based methodology also admits effective proof methods that are amenable for mechanized reasoning.

It is known that the (bi)simulation refinement and stuttering (bi)simulation refinement are compositional and support the stepwise refinement methodology [20, 24]. Moreover, the proof methods associated with them are local, *i.e.*, they only require reasoning about states and their successors. Hence, they are amenable to mechanized reasoning. However, to the best of our knowledge, it is not known if skipping refinement is compositional. Skipping refinement is a recently introduced notion of refinement for verifying the correctness of optimized implementations that can “execute faster” than their simple high-level specifications, *i.e.*, a step in the implementation can *skip* multiple steps in the specification. Examples of such systems include superscalar processors, concurrent and parallel systems and optimizing compilers. Two proof methods, *reduced well-founded skipping simulation* and *well-founded skipping simulation* have been introduced to reason about skipping refinement for the class of systems that exhibit bounded skipping [10]. These proof methods were used to verify the correctness of several systems that otherwise were difficult to automatically verify using current model-checkers and automated theorem provers. However, when skipping is unbounded, the proof methods in [10] require reachability analysis, and therefore are not amenable to automated reasoning. To motivate the need for alternative proof methods for effective reasoning, we consider the event processing system (EPS), discussed in [10].

1.1 Motivating Example

An abstract high-level specification, AEPS, of an event processing system is defined as follows. Let E be a set of *events* and V be a set of *state variables*. A *state* of AEPS is a triple $\langle t, Sch, St \rangle$, where t is a natural number denoting the current time; Sch is a set of pairs $\langle e, t_e \rangle$, where $e \in E$ is an event scheduled to be executed at time $t_e \geq t$; St is an assignment to state variables in V . The transition relation for the AEPS system is defined as follows. If at time t there is no $\langle e, t \rangle \in Sch$, *i.e.*, there is no event scheduled to be executed at time t , then t is incremented by 1. Otherwise, we (nondeterministically) choose and execute an event of the form $\langle e, t \rangle \in Sch$. The execution of an event may result in modifying St and also removing and adding a finite number of new pairs $\langle e', t' \rangle$ to Sch . We require that $t' > t$. Finally, execution involves removing the executed event $\langle e, t \rangle$ from Sch . Now consider, tEPS, an optimized implementation of AEPS. As before, a state is a triple $\langle t, Sch, St \rangle$. However, unlike the abstract system which just increments time by 1 when there are no events scheduled at the current time, the optimized system finds the earliest time in future an event is scheduled to execute. The transition relation of tEPS is defined as follows. An event (e, t_e)

with the minimum time is selected, t is updated to t_e and the event e is executed, as in the AEPS. Consider an execution of AEPS and tEPS in Fig. 1. (We only show the prefix of executions). Suppose at $t = 0$, Sch be $\{(e_1, 0)\}$. The execution of event e_1 add a new pair (e_2, k) to Sch , where k is a positive integer. AEPS at $t = 0$, executes the event e_1 , adds a new pair (e_2, k) to Sch , and updates t to 1. Since no events are scheduled to execute before $t = k$, the AEPS system repeatedly increments t by 1 until $t = k$. At $t = k$, it executes the event e_2 . At time $t = 0$, tEPS executes e_1 . The next event is scheduled to execute at time $t = k$; hence it updates in one step t to k . Next, in one step it executes the event e_2 . Note that tEPS runs faster than AEPS by *skipping* over abstract states when no event is scheduled for execution at the current time. If $k > 1$, the step from s_2 to s_3 in tEPS neither corresponds to stuttering nor to a single step of the AEPS. Therefore notions of refinement based on stuttering simulation and bisimulation cannot be used to show that tEPS refines AEPS.

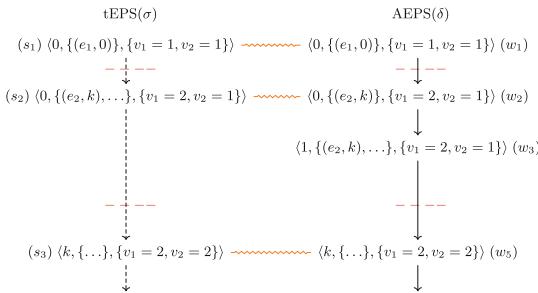


Fig. 1. Event simulation system

It was argued in [10] that skipping refinement is an appropriate notion of correctness that directly accounts for the skipping behavior exhibited by tEPS. Though, tEPS was used to motivate the need for a new notion of refinement, the proof methods proposed in [10] are not effective to prove the correctness of tEPS. This is because, execution of an event in tEPS may add new events that are scheduled to execute at an arbitrary time in future, *i.e.*, in general k in the above example execution is unbounded. Hence, the proof methods in [10] would require unbounded reachability analysis which often is problematic for automated verification tools. Even in the particular case when one can *a priori* determine an upper bound on k and unroll the transition relation, the proof methods in [10] are viable for mechanical reasoning only if the upper bound k is relatively small.

In this paper, we develop local proof methods to effectively analyze the correctness of optimized reactive systems using skipping refinement. These proof methods reduce global reasoning about infinite computations to local reasoning about states and their successor and are applicable even if the optimized implementation exhibits unbounded skipping. Moreover, we show that the proposed

proof methods are complete, *i.e.*, if a system \mathcal{M}_1 is a skipping refinement of \mathcal{M}_2 under a suitable refinement map, then we can always locally reason about them. We also develop an algebraic theory of skipping refinement. In particular, we show that skipping simulation is closed under relational composition. Thus, skipping refinement aligns with the stepwise refinement methodology. Finally, we illustrate the benefits of the theory of skipping refinement and the associated proof methods by verifying the correctness of optimized event processing systems in ACL2s [3].

2 Preliminaries

A transition system model of a reactive system captures the concept of a state, atomic transitions that modify state during the course of a computation, and what is observable in a state. Any system with a well defined operational semantics can be mapped to a labeled transition system.

Definition 1 Labeled Transition System. *A labeled transition system (TS) is a structure $\langle S, \rightarrow, L \rangle$, where S is a non-empty (possibly infinite) set of states, $\rightarrow \subseteq S \times S$, is a left-total transition relation (every state has a successor), and L is a labeling function whose domain is S .*

Notation: We first describe the notational conventions used in the paper. Function application is sometimes denoted by an infix dot “.” and is left-associative. The composition of relation R with itself i times (for $0 < i \leq \omega$) is denoted R^i ($\omega = \mathbb{N}$ and is the first infinite ordinal). Given a relation R and $1 < k \leq \omega$, $R^{<k}$ denotes $\bigcup_{1 \leq i < k} R^i$ and $R^{\geq k}$ denotes $\bigcup_{\omega > i \geq k} R^i$. Instead of $R^{<\omega}$ we often write the more common R^+ . \uplus denotes the disjoint union operator. Quantified expressions are written as $\langle Qx: r: t \rangle$, where Q is the quantifier (*e.g.*, $\exists, \forall, \min, \bigcup$), x is a bound variable, r is an expression that denotes the range of variable x (*true*, if omitted), and t is a term.

Let $\mathcal{M} = \langle S, \rightarrow, L \rangle$ be a transition system. An \mathcal{M} -path is a sequence of states such that for adjacent states, s and u , $s \rightarrow u$. The j^{th} state in an \mathcal{M} -path σ is denoted by $\sigma.j$. An \mathcal{M} -path σ starting at state s is a *fullpath*, denoted by $fp.\sigma.s$, if it is infinite. An \mathcal{M} -segment, $\langle v_1, \dots, v_k \rangle$, where $k \geq 1$ is a finite \mathcal{M} -path and is also denoted by \vec{v} . The length of an \mathcal{M} -segment \vec{v} is denoted by $|\vec{v}|$. Let INC be the set of strictly increasing sequences of natural numbers starting at 0. The i^{th} partition of a fullpath σ with respect to $\pi \in INC$, denoted by ${}^\pi\sigma^i$, is given by an \mathcal{M} -segment $\langle \sigma(\pi.i), \dots, \sigma(\pi(i+1)-1) \rangle$.

3 Theory of Skipping Refinement

In this section we first briefly recall the notion of skipping simulation as described in [10]. We then study the algebraic properties of skipping simulation and show that a theory of refinement based on it is compositional and therefore can be used in a stepwise refinement based verification methodology.

The definition of skipping simulation is based on the notion of *matching*. Informally, a fullpath σ matches a fullpath δ under the relation B iff the fullpaths can be partitioned in to non-empty, finite segments such that all elements in a segment of σ are related to the first element in the corresponding segment of δ .

Definition 2 smatch [10]. *Let $\mathcal{M} = \langle S, \rightarrow, L \rangle$ be a transition system, σ, δ be fullpaths in \mathcal{M} . For $\pi, \xi \in INC$ and binary relation $B \subseteq S \times S$, we define*

$$\begin{aligned} scorr(B, \sigma, \pi, \delta, \xi) &\equiv \langle \forall i \in \omega :: \langle \forall s \in {}^\pi\sigma^i :: sB\delta(\xi.i) \rangle \rangle \text{ and} \\ smatch(B, \sigma, \delta) &\equiv \langle \exists \pi, \xi \in INC :: scorr(B, \sigma, \pi, \delta, \xi) \rangle. \end{aligned}$$

Figure 1 illustrates the notion of matching using our running example: σ is the fullpath of the concrete system and δ is a fullpath of the abstract system. (The figure only shows the prefix of the fullpaths). The other parameter for matching is the relation B , which is just the identity function. In order to show that $smatch(B, \sigma, \delta)$ holds, we have to find $\pi, \xi \in INC$ satisfying the definition. In Fig. 1, we separate the partitions induced by our choice for π, ξ using $\dashv\dashv$ and connect elements related by B with \curvearrowright . Since all elements of a σ partition are related to the first element of the corresponding δ partition, $scorr(B, \sigma, \pi, \delta, \xi)$ holds, therefore, $smatch(B, \sigma, \delta)$ holds.

Using the notion of matching, skipping simulation is defined as follows. Notice that skipping simulation is defined using a single transition system; it is easy to lift the notion defined on a single transition system to one that relates two transition systems by taking the disjoint union of the transition systems.

Definition 3 Skipping Simulation (SKS). *$B \subseteq S \times S$ is a skipping simulation on a TS $\mathcal{M} = \langle S, \rightarrow, L \rangle$ iff for all s, w such that sBw , both of the following hold.*

- (SKS1) $L.s = L.w$
- (SKS2) $\langle \forall \sigma: fp.\sigma.s: \langle \exists \delta: fp.\delta.w: smatch(B, \sigma, \delta) \rangle \rangle$

Theorem 1. *Let \mathcal{M} be a TS. If B is a stuttering simulation (STS) on \mathcal{M} then B is an SKS on \mathcal{M} .*

Proof: Follows directly from the definitions of SKS and STS [18]. □

3.1 Algebraic Properties

We now study the algebraic properties of SKS. We show that it is closed under arbitrary union. We also show that SKS is closed under relational composition. The later property is particularly useful since it enables us to use stepwise refinement and to modularly analyze the correctness of complex systems.

Lemma 1. *Let \mathcal{M} be a TS and \mathcal{C} be a set of SKS's on \mathcal{M} . Then $G = \langle \cup B : B \in \mathcal{C} : B \rangle$ is an SKS on \mathcal{M} .*

Corollary 1. *For any TS \mathcal{M} , there is a greatest SKS on \mathcal{M} .*

Lemma 2. *SKS are not closed under negation and intersection.*

The following lemma shows that skipping simulation is closed under relational composition.

Lemma 3. *Let \mathcal{M} be a TS. If P and Q are SKS's on \mathcal{M} , then $R = P; Q$ is an SKS on \mathcal{M} .*

Proof: To show that R is an SKS on $\mathcal{M} = \langle S, \rightarrow, L \rangle$, we show that for any $s, w \in S$ such that sRw , SKS1 and SKS2 hold. Let $s, w \in S$ and sRw . From the definition of R , there exists $x \in S$ such that sPx and xQw . Since P and Q are SKS's on \mathcal{M} , $L.s = L.x = L.w$, hence, SKS1 holds for R .

To prove that SKS2 holds for R , consider a fullpath σ starting at s . Since P and Q are SKSs on \mathcal{M} , there is a fullpath τ in \mathcal{M} starting at x , a fullpath δ in \mathcal{M} starting at w and $\alpha, \beta, \theta, \gamma \in INC$ such that $scorr(P, \sigma, \alpha, \tau, \beta)$ and $scorr(Q, \tau, \theta, \delta, \gamma)$ hold. We use the fullpath δ as a witness and define $\pi, \xi \in INC$ such that $scorr(R, \sigma, \pi, \delta, \xi)$ holds.

We define a function, r , that given i , corresponding to the index of a partition of τ under β , returns the index of the partition of τ under θ in which the first element of τ 's i^{th} partition under β resides. $r.i = j$ iff $\theta.j \leq \beta.i < \theta(j+1)$. Note that r is indeed a function, as every element of τ resides in exactly one partition of θ . Also, since there is a correspondence between the partitions of α and β , (by $scorr(P, \sigma, \alpha, \tau, \beta)$), we can apply r to indices of partitions of σ under α to find where the first element of the corresponding β partition resides. Note that r is non-decreasing: $a < b \Rightarrow r.a \leq r.b$.

We define $\pi\alpha \in INC$, a strictly increasing sequence that will allow us to merge adjacent partitions in α as needed to define the strictly increasing sequence π on σ used to prove SKS2. Partitions in π will consist of one or more α partitions. Given i , corresponding to the index of a partition of σ under π , the function $\pi\alpha$ returns the index of the corresponding partition of σ under α .

$$\pi\alpha(0) = 0$$

$$\pi\alpha(i) = \min j \in \omega \text{ s.t. } |\{k : 0 < k \leq j \wedge r.k \neq r(k-1)\}| = i$$

Note that $\pi\alpha$ is an increasing function, i.e., $a < b \Rightarrow \pi\alpha(a) < \pi\alpha(b)$. We now define π as follows.

$$\pi.i = \alpha(\pi\alpha.i)$$

There is an important relationship between r and $\pi\alpha$

$$r(\pi\alpha.i) = \dots = r(\pi\alpha(i+1)-1)$$

That is, for all α partitions that are in the same π partition, the initial states of the corresponding β partitions are in the same θ partition.

We define ξ as follows: $\xi.i = \gamma(r(\pi\alpha.i))$.

We are now ready to prove SKS2. Let $s \in {}^\pi\sigma^i$. We show that $sR\delta(\xi.i)$. By the definition of π , we have

$$s \in {}^\alpha\sigma^{\pi\alpha.i} \vee \cdots \vee s \in {}^\alpha\sigma^{\pi\alpha(i+1)-1}$$

Hence,

$$sP\tau(\beta(\pi\alpha.i)) \vee \cdots \vee sP\tau(\beta(\pi\alpha(i+1)-1))$$

Note that by the definition of r (apply r to $\pi\alpha.i$):

$$\theta(r(\pi\alpha.i)) \leq \beta(\pi\alpha.i) < \theta(r(\pi\alpha.i) + 1)$$

Hence,

$$\tau(\beta(\pi\alpha.i))Q\delta(\gamma(r(\pi\alpha.i))) \vee \cdots \vee \tau(\beta(\pi\alpha(i+1)-1))Q\delta(\gamma(r(\pi\alpha(i+1)-1)))$$

By the definition of ξ and the relationship between r and $\pi\alpha$ described above, we simplify the above formula as follows.

$$\tau(\beta(\pi\alpha.i))Q\delta(\xi.i) \vee \cdots \vee \tau(\beta(\pi\alpha(i+1)-1))Q\delta(\xi.i)$$

Therefore, by the definition of R , we have that $sR\delta(\xi.i)$ holds. \square

Theorem 2. *The reflexive transitive closure of an SKS is an SKS.*

Theorem 3. *Given a TS \mathcal{M} , the greatest SKS on \mathcal{M} is a preorder.*

Proof. Let G be the greatest SKS on \mathcal{M} . From Theorem 2, G^* is an SKS. Hence $G^* \subseteq G$. Furthermore, since $G \subseteq G^*$, we have that $G = G^*$, i.e., G is reflexive and transitive. \square

3.2 Skipping Refinement

We now recall the notion of skipping refinement [10]. We use skipping simulation, a notion defined in terms of a single transition system, to define skipping refinement, a notion that relates *two* transition systems: an *abstract* transition system and a *concrete* transition system. Informally, if a concrete system is a skipping refinement of an abstract system, then its observable behaviors are also behaviors of the abstract system, modulo skipping (which includes stuttering). The notion is parameterized by a *refinement map*, a function that maps concrete states to their corresponding abstract states. A refinement map along with a labeling function determines what is observable at a concrete state.

Definition 4 Skipping Refinement. Let $\mathcal{M}_A = \langle S_A, \xrightarrow{A}, L_A \rangle$ and $\mathcal{M}_C = \langle S_C, \xrightarrow{C}, L_C \rangle$ be transition systems and let $r : S_C \rightarrow S_A$ be a refinement map. We say \mathcal{M}_C is a skipping refinement of \mathcal{M}_A with respect to r , written $\mathcal{M}_C \lesssim_r \mathcal{M}_A$, if there exists a binary relation B such that all of the following hold.

1. $\langle \forall s \in S_C :: sBr.s \rangle$ and
2. B is an SKS on $\langle S_C \uplus S_A, \xrightarrow{C} \uplus \xrightarrow{A}, \mathcal{L} \rangle$ where $\mathcal{L}.s = L_A(s)$ for $s \in S_A$, and $\mathcal{L}.s = L_A(r.s)$ for $s \in S_C$.

Next, we use the property that skipping simulation is closed under relational composition to show that skipping refinement supports modular reasoning using a stepwise refinement approach. In order to verify that a low-level complex implementation \mathcal{M}_C refines a simple high-level abstract specification \mathcal{M}_A one proceeds as follows: starting with \mathcal{M}_A define a sequence of intermediate systems leading to the final complex implementation \mathcal{M}_C . Any two successive systems in the sequence differ only in relatively few aspects of their behavior. We then show that, at each step in the sequence, the system at the current step is a refinement of the previous one. Since at each step, the verification effort is focused only on the few differences in behavior between two systems under consideration, proof obligations are simpler than the monolithic proof. Note that this methodology is orthogonal to (horizontal) modular reasoning that infers the correctness of a system from the correctness of its sub-components.

Theorem 4. Let $\mathcal{M}_1 = \langle S_1, \xrightarrow{1}, L_1 \rangle$, $\mathcal{M}_2 = \langle S_2, \xrightarrow{2}, L_2 \rangle$, and $\mathcal{M}_3 = \langle S_3, \xrightarrow{3}, L_3 \rangle$ be TSs, $p : S_1 \rightarrow S_2$ and $r : S_2 \rightarrow S_3$. If $\mathcal{M}_1 \lesssim_p \mathcal{M}_2$ and $\mathcal{M}_2 \lesssim_r \mathcal{M}_3$, then $\mathcal{M}_1 \lesssim_{p;r} \mathcal{M}_3$.

Proof: Since $\mathcal{M}_1 \lesssim_p \mathcal{M}_2$, we have an SKS, say A , such that $\langle \forall s \in S_1 :: sA(p.s) \rangle$. Furthermore, without loss of generality we can assume that $A \subseteq S_1 \times S_2$. Similarly, since $\mathcal{M}_2 \lesssim_r \mathcal{M}_3$, we have an SKS, say B , such that $\langle \forall s \in S_2 :: sB(r.s) \rangle$ and $B \subseteq S_2 \times S_3$. Define $C = A; B$. Then we have that $C \subseteq S_1 \times S_3$ and $\langle \forall s \in S_1 :: sCr(p.s) \rangle$. Also, from Theorem 2, C is an SKS on $\langle S_1 \uplus S_3, \xrightarrow{1} \uplus \xrightarrow{3}, \mathcal{L} \rangle$, where $\mathcal{L}.s = L_3(s)$ if $s \in S_3$ else $\mathcal{L}.s = L_3(r(p.s))$.

Formally, to establish that a complex low-level implementation \mathcal{M}_C refines a simple high-level abstract specification \mathcal{M}_A , one defines intermediate systems $\mathcal{M}_1, \dots, \mathcal{M}_n$, where $n \geq 1$ and establishes the following: $\mathcal{M}_C = \mathcal{M}_0 \lesssim_{r_0} \mathcal{M}_1 \lesssim_{r_1} \dots \lesssim_{r_{n-1}} \mathcal{M}_n = \mathcal{M}_A$. Then from Theorem 4, we have that $\mathcal{M}_C \lesssim_r \mathcal{M}_A$, where $r = r_0; r_1; \dots; r_{n-1}$. We illustrate the utility of this approach in Sect. 5 by proving the correctness of an optimized event processing systems.

Theorem 5. Let $\mathcal{M} = \langle S, \rightarrow, L \rangle$ be a TS. Let $\mathcal{M}' = \langle S', \rightarrow^{'}, L' \rangle$ where $S' \subseteq S$, $\rightarrow^{'} \subseteq S' \times S'$, $\rightarrow^{'} is a left-total subset of \rightarrow^+ , and $L' = L|_{S'}$. Then $\mathcal{M}' \lesssim_I \mathcal{M}$, where I is the identity function on S' .$

Corollary 2. Let $\mathcal{M}_C = \langle S_C, \xrightarrow{C}, L_C \rangle$ and $\mathcal{M}_A = \langle S_A, \xrightarrow{A}, L_A \rangle$ be TSs, $r : S_C \rightarrow S_A$ be a refinement map. Let $\mathcal{M}'_C = \langle S'_C, \xrightarrow{C'}, L'_C \rangle$ where $S'_C \subseteq S_C$, $\xrightarrow{C'}$ is a left-total subset of \xrightarrow{C}^+ , and $L'_C = L_C|_{S'_C}$. If $\mathcal{M}_C \lesssim_r \mathcal{M}_A$ then $\mathcal{M}'_C \lesssim_{r'} \mathcal{M}_A$, where r' is $r|_{S'_C}$.

We now illustrate the usefulness of the theory of skipping refinement using our running example of event processing systems. Consider MPEPS, that uses

a priority queue to find a non-empty set of events (say E_t) scheduled to execute at the current time and executes them. We allow the priority queue in MPEPS to be deterministic or nondeterministic. For example, the priority queue may deterministically select a single event in E_t to execute, or based on considerations such as resource utilization it may execute some subset of events in E_t in a single step. When reasoning about the correctness of MPEPS, one thing to notice is that there is a difference in the data structures used in the two systems: MPEPS uses a priority queue to effectively find the next set of events to execute in the scheduler, while AEPS uses a simple abstract set representation for the scheduler. Another thing to notice is that MPEPS can “execute faster” than AEPS in two ways: it can increment time by more than 1 and it can execute more than one event in a single step. The theory of skipping refinement developed in this paper enables us to separate out these concerns and apply a stepwise refinement approach to effectively analyse MPEPS.

First, we account for the difference in the data structures between MPEPS and AEPS. Towards this we define an intermediate system MEPS that is identical to MPEPS except that the scheduler in MEPS is now represented as a set of event-time pairs. Under a refinement map, say p , that extracts the set of event-time pairs in the priority queue of MPEPS, a step in MPEPS can be matched by a step in MEPS. Hence, $\text{MPEPS} \lesssim_p \text{MEPS}$. Next we account for the difference between MEPS and AEPS in the number of events the two systems may execute in a single step. Towards this, observe that the state space of MEPS and tEPS are equal and the transition relation of MEPS is a left-total subset of the transitive closure of the transition relation of tEPS. Hence, from Theorem 5, we infer that MPEPS is a skipping refinement of tEPS using the identity function, say I_1 , as the refinement map, i.e., $\text{MEPS} \lesssim_{I_1} \text{tEPS}$. Next observe that the state spaces of tEPS and AEPS are equal and the transition relation of tEPS is a left-total subset of the transitive closure of the transition relation of AEPS. Hence, from Theorem 5, tEPS is a skipping refinement of AEPS using the identity function, say I_2 , as the refinement map, i.e., $\text{tEPS} \lesssim_{I_2} \text{AEPS}$. Finally, from the transitivity of skipping refinement (Theorem 4), we conclude that $\text{MPEPS} \lesssim_{p'} \text{AEPS}$, where $p' = p; I_1; I_2$.

4 Mechanised Reasoning

To prove that a transition system \mathcal{M}_C is a skipping refinement of a transition system \mathcal{M}_A using Definition 3, requires us to show that for any fullpath from \mathcal{M}_C we can find a matching fullpath from \mathcal{M}_A . However, reasoning about existence of infinite sequences can be problematic using automated tools. In this section, we develop sound and complete local proof methods that are applicable even if a system exhibits unbounded skipping. We first briefly present the proof methods, reduced well-founded skipping and well-founded skipping simulation, developed in [10].

Definition 5 Reduced Well-founded Skipping [10]. $B \subseteq S \times S$ is a reduced well-founded skipping relation on $\text{TS } \mathcal{M} = \langle S, \rightarrow, L \rangle$ iff:

(RWFSK1) $\langle \forall s, w \in S : sBw : L.s = L.w \rangle$

(RWFSK2) There exists a function, $\text{rankt} : S \times S \rightarrow W$, such that $\langle W, \prec \rangle$ is well-founded and

$$\begin{aligned} & \langle \forall s, u, w \in S : s \rightarrow u \wedge sBw : \\ & \quad (a) (uBw \wedge \text{rankt}(u, w) \prec \text{rankt}(s, w)) \vee \\ & \quad (b) \langle \exists v : w \rightarrow^+ v : uBv \rangle \rangle \end{aligned}$$

Definition 6 Well-founded Skipping [10]. $B \subseteq S \times S$ is a well-founded skipping relation on TS $\mathcal{M} = \langle S, \rightarrow, L \rangle$ iff:

(WFSK1) $\langle \forall s, w \in S : sBw : L.s = L.w \rangle$

(WFSK2) There exist functions, $\text{rankt} : S \times S \rightarrow W$, $\text{rankl} : S \times S \times S \rightarrow \omega$, such that $\langle W, \prec \rangle$ is well-founded and

$$\begin{aligned} & \langle \forall s, u, w \in S : s \rightarrow u \wedge sBw : \\ & \quad (a) \langle \exists v : w \rightarrow v : uBv \rangle \vee \\ & \quad (b) (uBw \wedge \text{rankt}(u, w) \prec \text{rankt}(s, w)) \vee \\ & \quad (c) \langle \exists v : w \rightarrow v : sBv \wedge \text{rankl}(v, s, u) < \text{rankl}(w, s, u) \rangle \vee \\ & \quad (d) \langle \exists v : w \rightarrow^{\geq 2} v : uBv \rangle \rangle \end{aligned}$$

Theorem 6 [10]. Let $\mathcal{M} = \langle S, \rightarrow, L \rangle$ be a TS and $B \subseteq S \times S$. The following statements are equivalent

- (i) B is a SKS on \mathcal{M} ;
- (ii) B is a WFSK on \mathcal{M} ;
- (iii) B is a RWFSK on \mathcal{M} .

Recall the event processing systems AEPS and tEPS described in Sect. 1.1. When no events are scheduled to execute at a given time, say t , tEPS increments time t to the earliest time in future, say $k > t$, at which an event is scheduled for execution. Execution of an event can add an event that is scheduled to be executed at an arbitrary time in future. Therefore, we cannot apriori determine an upper-bound on k . Using any of the above two proof-methods to reason about skipping refinement would require unbounded reachability analysis (conditions RWFSK2b and WFSK2d), often difficult for automated verification tools. To redress the situation, we develop two new proof methods of SKS; both require only local reasoning about steps and their successors.

Definition 7 Reduced Local Well-founded Skipping. $B \subseteq S \times S$ is a local well-founded skipping relation on TS $\mathcal{M} = \langle S, \rightarrow, L \rangle$ iff:

(RLWFSK1) $\langle \forall s, w \in S : sBw : L.s = L.w \rangle$

(RLWFSK2) There exist functions, $\text{rankt} : S \times S \longrightarrow W$, $\text{rankls} : S \times S \longrightarrow \omega$ such that $\langle W, \prec \rangle$ is well founded, and, a binary relation $\mathcal{O} \subseteq S \times S$

such that

$$\langle \forall s, u, w \in S : sBw \wedge s \rightarrow u : \\ (a) (uBw \wedge \text{rankt}(u, w) \prec \text{rankt}(s, w)) \vee \\ (b) (\exists v : w \rightarrow v : u\mathcal{O}v) \rangle$$

and

$$\langle \forall x, y \in S : x\mathcal{O}y : \\ (c) xBy \vee \\ (d) (\exists z : y \rightarrow z : x\mathcal{O}z \wedge \text{rankls}(z, x) < \text{rankls}(y, x)) \rangle$$

Observe that to prove that a relation is an RLWFSK on a transition system, it is sufficient to reason about single steps of the transition system. Also, note that RLWFSK does not differentiate between skipping and stuttering on the right. This is based on an earlier observation that skipping subsumes stuttering. We used this observation to simplify the definition. However, it can often be useful to differentiate between skipping and stuttering. Next we define local well-founded skipping simulation (LWFSK), a characterization of skipping simulation that separates reasoning about skipping and stuttering on the right (Fig. 2).

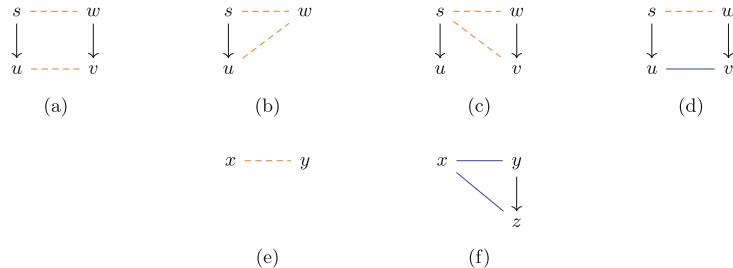


Fig. 2. Local well-founded skipping simulation (orange line indicates the states are related by B and blue line indicate the states are related by \mathcal{O}) (Color figure online)

Definition 8 Local Well-founded Skipping. $B \subseteq S \times S$ is a local well-founded skipping relation on $TS \mathcal{M} = \langle S, \rightarrow, L \rangle$ iff:

$$(LWFSK1) \langle \forall s, w \in S : sBw : L.s = L.w \rangle$$

$$(LWFSK2) \text{ There exist functions, } \text{rankt} : S \times S \longrightarrow W, \text{ rankl} : S \times S \times S \longrightarrow \omega, \text{ and } \text{rankls} : S \times S \longrightarrow \omega \text{ such that } \langle W, \prec \rangle \text{ is well founded, and, a}$$

binary relation $\mathcal{O} \subseteq S \times S$ such that

- $$\langle \forall s, u, w \in S : sBw \wedge s \rightarrow u : \begin{aligned} & (a) \langle \exists v : w \rightarrow v : uBv \rangle \vee \\ & (b) (uBw \wedge rankt(u, w) \prec rankt(s, w)) \vee \\ & (c) \langle \exists v : w \rightarrow v : sBv \wedge rankl(v, s, u) < rankl(w, s, u) \rangle \vee \\ & (d) \langle \exists v : w \rightarrow v : u\mathcal{O}v \rangle \end{aligned} \rangle$$

and

- $$\langle \forall x, y \in S : x\mathcal{O}y : \begin{aligned} & (e) xBy \vee \\ & (f) \langle \exists z : y \rightarrow z : x\mathcal{O}z \wedge rankls(z, x) < rankls(y, x) \rangle \end{aligned} \rangle$$

Like RLWFSK, to prove that a relation is a LWFSK, reasoning about single steps of the transition system suffices. However, LWFSK2b accounts for stuttering on the right, and LWFSK2d along with LWFSK2e and LWFSK2f accounts for skipping on the right. Also observe that states related by \mathcal{O} are not required to be labeled identically and may have no observable relationship to the states related by B .

Soundness and Completeness. We next show that RLWFSK and LWFSK in fact completely characterize skipping simulation, *i.e.*, RLWFSK and LWFSK are sound and complete proof rules. Thus if a concrete system \mathcal{M}_C is a skipping refinement of \mathcal{M}_A , one can always effectively reason about it using RLWFSK and LWFSK.

Theorem 7. *Let $\mathcal{M} = \langle S, \rightarrow, L \rangle$ be a transition system and $B \subseteq S \times S$. The following statements are equivalent:*

- (i) *B is an SKS on \mathcal{M} ;*
- (ii) *B is a WFSK on \mathcal{M} ;*
- (iii) *B is an RWFSK on \mathcal{M} ;*
- (iv) *B is an RLWFSK on \mathcal{M} ;*
- (v) *B is a LWFSK on \mathcal{M} ;*

Proof: The equivalence of (i), (ii) and (iii) follows from Theorem 6. That (iv) implies (v) follows from the simple observation that RLWFSK2 implies LWFSK2. To complete the proof, we prove the following two implications. We prove below that (v) implies (ii) in Lemma 4 and that (iii) implies (iv) in Lemma 5. \square

Lemma 4. *If B is a LWFSK on \mathcal{M} , then B is a WFSK on \mathcal{M} .*

Proof. Let B be a LWFSK on \mathcal{M} . WFSK1 follows directly from LWFSK1. Let $rankt$, $rankl$, and $rankls$ be functions, and \mathcal{O} be a binary relation such that LWFSK2 holds. To show that WFSK2 holds, we use the same $rankt$ and $rankl$ functions and let $s, u, w \in S$ and $s \rightarrow u$ and sBw . LWFSK2a, LWFSK2b and

LWFSK2c are equivalent to WFSK2a, WFSK2b and WFSK2c, respectively, so we show that if only LWFSK2d holds, then WFSK2d holds. Since LWFSK2d holds, there is a successor v of w such that $u\mathcal{O}v$. Since $u\mathcal{O}v$ holds, either LWFSK2e or LWFSK2f must hold between u and v . However, since LWFSK2a does not hold, LWFSK2e cannot hold and LWFSK2f must hold, *i.e.*, there exists a successor v' of v such that $u\mathcal{O}v' \wedge rankls(v', u) < rankls(v, u)$. So, we need a path of at least 2 steps from w to satisfy the universally quantified constraint on \mathcal{O} . Let us consider an arbitrary path, δ , such that $\delta.0 = w$, $\delta.1 = v$, $\delta.2 = v'$, $u\mathcal{O}\delta.i$, LWFSK2e does not hold between u and $\delta.i$ for $i \geq 1$, and $rankls(\delta.(i+1), u) < rankls(\delta.i, u)$. Notice that any such path must be finite because $rankls$ is well founded. Hence, δ is a finite path and there exists a $k \geq 2$ such that LWFSK2e holds between u and $\delta.k$. Therefore, WFSK2d holds, *i.e.*, there is a state in δ reachable from w in two or more steps which is related to u by B . \square

Lemma 5. *If B is RWFSK on \mathcal{M} , then B is an RLWFSK on \mathcal{M} .*

Proof. Let B be an RWFSK on \mathcal{M} . RLWFSK1 follows directly from RWFSK1. To show that RLWFSK2 holds, we use any $rankt$ function that can be used to show that RWFSK2 holds. We define \mathcal{O} as follows.

$$\mathcal{O} = \{(u, v) : \langle \exists z : v \rightarrow^+ z : uBz \rangle\}$$

We define $rankls(u, v)$ to be the minimal length of a \mathcal{M} -segment that starts at v and ends at a state, say z , such that uBz , if such a segment exists and 0 otherwise. Let $s, u, w \in S$, sBw and $s \rightarrow u$. If RWFSK2a holds between s, u , and w , then RLWFSK2a also holds. Next, suppose that RWFSK2a does not hold but RWFSK2b holds, *i.e.*, there is an \mathcal{M} -segment $\langle w, a, \dots, v \rangle$ such that uBv ; therefore, $u\mathcal{O}a$ and RLWFSK2b holds.

To finish the proof, we show that \mathcal{O} and $rankls$ satisfy the constraints imposed by the second conjunct in RLWFSK2. Let $x, y \in S$, $x\mathcal{O}y$ and $x \not\mathcal{B} y$. From the definition of \mathcal{O} , we have that there is an \mathcal{M} -segment from y to a state related to x by B ; let \vec{y} be such a segment of minimal length. From definition of $rankls$, we have $rankls(y, x) = |\vec{y}|$. Observe that y cannot be the last state of \vec{y} and $|\vec{y}| \geq 2$. This is because the last state in \vec{y} must be related to x by B , but from the assumption we know that $x \not\mathcal{B} y$. Let y' be a successor of y in \vec{y} . Clearly, $x\mathcal{O}y'$; therefore, $rankls(y', x) < |\vec{y}| - 1$, since the length of a minimal \mathcal{M} -segment from y' to a state related to x by B , must be less or equal to $|\vec{y}| - 1$. \square

5 Case Study (Event Processing System)

In this section, we analyze the correctness of an optimized event processing system (PEPS) that uses a *priority queue* to find an event scheduled to execute at any given time. We show that PEPS refines AEPS, a simple event processing system described in Sect. 1. Our goal is to illustrate the benefits of the theory of skipping refinement and the associated local proof methods developed in the

paper. We use ACL2s [3], an interactive theorem prover, to define the operational semantics of the systems and mechanize a proof of its correctness.

Operational Semantics of PEPS: A state of PEPS system is a triple $\langle \text{tm}, \text{otevs}, \text{mem} \rangle$, where tm is a natural number denoting current time, otevs is a set of timed-event pairs denoting the scheduler that is ordered with respect to a total order $\text{te}-<$ on timed-event pairs, and mem is a collection of variable-integer pairs denoting the shared memory. The transition function of PEPS is defined as follows: if there are no events in otevs , then PEPS just increments the current time by 1. Otherwise, it picks the first timed-event pair, say $\langle e, t \rangle$ in otevs , executes it and updates the time to t . The execution of an event may result in adding new timed-events to the scheduler, removing existing timed-events from the scheduler and updating the memory. Finally, the executed timed-event is removed from the scheduler. This is a simple, generic model of an event processing system. Notice that the ability to remove events can be used to specify systems with preemption [23]: an event scheduled to execute at some future time may be canceled (and possibly rescheduled to be executed at a different time in future) as a result of the execution of an event that preempts it. Notice that, for a given total order, PEPS is a deterministic system.

The execution of an event is modeled using three constrained functions that take as input an event, ev , a time, t , and a memory, mem : `step-events-add` returns the set of new timed-event pairs to add to the scheduler; `step-events-rm` returns the set of timed-event pairs to remove from the scheduler; and `step-memory` returns a memory updated as specified by the event. We place minimal constraints on these functions. For example, we only require that `step-events-add` returns a set of event-time pairs of the form $\langle e, t_e \rangle$ where t_e is greater than the current time t . The constrained functions are defined using the `encapsulate` construct in ACL2 and can be instantiated with any executable definitions that satisfy these constraints without affecting the proof of correctness of PEPS. Moreover, note that the particular choice of the total order on timed-event pairs is irrelevant to the proof of correctness of PEPS.

Stepwise Refinement: We show that PEPS refines AEPS using a stepwise refinement approach: first we define an intermediate system HPEPS obtained by augmenting PEPS with history information and show that PEPS is a simulation refinement of HPEPS. Second, we show that HPEPS is a skipping refinement of AEPS. Finally, we appeal to Theorems 1 and 4 to infer that PEPS refines AEPS. Note that the compositionality of skipping refinement enables us to decompose the proof into a sequence of refinement proofs, each of which is simpler. Moreover, the history information in HPEPS is helpful in defining the witnessing binary relation and the rank function required to prove skipping refinement.

An HPEPS state is a four-tuple $\langle \text{tm}, \text{otevs}, \text{mem}, h \rangle$, where tm , otevs , mem are respectively the current time, an ordered set of timed events and a collection of variable-integer pairs, and h is the history information. The history information h consists of a Boolean variable `valid`, time tm , and an ordered set of timed-event pairs otevs and the memory mem . Intuitively, h records the state preceding the

current state. The transition function HPEPS is same as the transition function of PEPS except that HPEPS also records the history in \mathbf{h} .

PEPS Refines HPEPS: Observe that, modulo the history information, a step of PEPS directly corresponds to a step of HPEPS, *i.e.*, PEPS is a bisimulation refinement of HPEPS under a refinement map that projects a PEPS state $\langle tm, otevs, mem \rangle$ to the HPEPS state $\langle tm, otevs, mem, h \rangle$ where the `valid` component of \mathbf{h} is set to false. But we only prove that it is a simulation refinement, because, from Theorem 1, it suffices to establish that PEPS is a skipping refinement of HPEPS. The proofs primarily require showing that two sets of ordered timed-events that are set equivalent are in fact equal and that adding and removing equivalent sets of timed-event from equal schedulers results in equal schedulers.

HPEPS Refines AEPS: Next we show that HPEPS is a skipping refinement of AEPS under the refinement map R , a function that simply projects an HPEPS state to an AEPS state. To show that HPEPS is a skipping refinement of AEPS under the refinement map R , from Definition 4, we must show as witness a binary relation B that satisfies the two conditions. Let $B = \{(s, R.s) : s \text{ is an HPEPS state}\}$. To establish that B is an SKS on the disjoint union of HPEPS and AEPS, we have a choice of four proof-methods (Sect. 4). Recall that execution of an event can add a new event scheduled to be executed at an arbitrary time in the future. As a result, if we were to use WFSK or RWFSK, the proof obligations from conditions WFSK2d (Definition 5) and RWFSK2b (Definition 6) would require unbounded reachability analysis, something that typically places a big burden on verification tools and their users. In contrast, the proof obligations to establish RLWFSK are local and only require reasoning about states and their successors, which significantly reduces the proof complexity.

RLWFSK1 holds trivially. To prove that RLWFSK2 holds we define a binary relation \mathcal{O} and a rank function $rankls$ and show that they satisfy the two universally quantified formulas in RLWFSK2. Moreover, since HPEPS does not stutter we ignore RLWFSK2a, and that is why we do not define $rankt$. Finally, our proof obligation is: for all HPEPS s, u and AEPS state w such that $s \rightarrow u$ and sBw holds, there exists a AEPS state v such that $w \rightarrow v$ and $u\mathcal{O}v$ holds.

Verification Effort: We used the `defdata` framework in ACL2s, to specify the data definitions for the three systems and the `definec` construct to introduce function definitions along with their input-contracts (pre-conditions) and output-contracts (post-conditions). In addition to admitting a data definition, `defdata` proves several theorems about the functions that are extremely helpful in automatically discharging type-like proof obligations. We also developed a library to concisely describe functions using higher-order constructs like `map` and `reduce`, which made some of the definitions clearer. ACL2s supports first-order quantifiers via the `defun-sk` construct, which essentially amounts to the use of Hilbert's choice operator. We use `defun-sk` to model the transition relation for AEPS (a non-deterministic system) and to specify the proof obligations for proving that HPEPS refines AEPS. However, support for automated reasoning

about quantifiers is limited in ACL2. Therefore, we use the domain knowledge, when possible (*e.g.*, a system is deterministic), to eliminate quantifiers in the proof obligations and provide explicit witnesses for existential quantifiers.

The proof makes essential use of several libraries available in ACL2 for reasoning about lists and sets. In addition, we prove a collection of additional lemmas that can be roughly categorized into four categories. First, we have a collection of lemmas to prove the input-output contracts of the functions. Second, we have a collection of lemmas to show that operations on the schedulers in the three systems preserve various invariants, *e.g.*, that any timed-event in the scheduler is scheduled to execute at a time greater or equal to the current time. Third, we have a collection of lemmas to show that inserting and removing two equivalent sets of timed-events from a scheduler results in an equivalent scheduler. And fourth, we have a collection of lemmas to show that two schedulers are equivalent *iff* they are set equal. The above lemmas are used to establish a relationship between priority queues, a data structure used by the implementation system, and sets, the corresponding data structure used in the specification system. The behavioral difference between the two systems is accounted for by the notion of skipping refinement. This separation significantly eases understanding as well as mechanical reasoning about the correctness of reactive systems. We have 8 top-level proof obligations and a few dozen supporting lemmas. The entire proof takes about 120 s on a machine with 2.2 GHz Intel Core i7 with 16 GB main memory.

6 Related Work

Several notions of correctness have been proposed in the literature and their properties been widely studied [2, 5, 11, 16, 17]. In this paper, we develop a theory of skipping refinement to effectively prove the correctness of optimized reactive systems using automated verification tools. These results establish skipping refinement on par with notions of refinement based on (bi)simulation [22] and stuttering (bi)simulation [20, 24], in the sense that skipping refinement is (1) compositional and (2) admits local proofs methods. Together the two properties have been instrumental in significantly reducing the proof complexity in verification of large and complex systems. We developed the theory of skipping refinement using a generic model of transition systems and place no restrictions on the state space size or the branching factor of the transition system. Any system with a well-defined operational semantics can be mapped to a labeled transition system. Moreover, the local proof methods are sound and complete, *i.e.*, if an implementation is a skipping refinement of the specification, we can always use the local proof methods to effectively reason about it.

Refinement-based methodologies have been successfully used to verify the correctness of several realistic hardware and software systems. In [13], several complex concurrent programs were verified using a stepwise refinement methodology. In addition, Kragl and Qadeer [13] also develop a compact representation to facilitate the description of programs at different levels of abstraction and associated refinement proofs. Several back-end compiler transformations are proved

correct in Compcert [15] using simulation refinement. In [25], several compiler transformations were verified using stuttering refinement and associated local proof methods. Recently, refinement-based methodology has also been applied to verify the correctness of practical distributed systems [8] and a general-purpose operating system microkernel [12]. The full verification of CertiKOS [6,7], an OS kernel, is based on the notion of simulation refinement. Refinement based approaches have also been extensively used to verify microprocessor designs [1,9,19,21,26]. Skipping refinement was used to verify the correctness of optimized memory controllers and a JVM-inspired stack machine [10].

7 Conclusion and Future Work

In this paper, we developed the theory of skipping refinement. Skipping refinement is designed to reason about the correctness of optimized reactive systems, a class of systems where a single transition in a concrete low-level implementation may correspond to a sequence of observable steps in the corresponding abstract high-level specification. Examples of such systems include optimizing compilers, concurrent and parallel systems and superscalar processors. We developed sound and complete proof methods that reduce global reasoning about infinite computations of such systems to local reasoning about states and their successors. We also showed that the skipping simulation is closed under composition and therefore is amenable to modular reasoning using a stepwise refinement approach. We experimentally validated our results by analyzing the correctness of an optimized event-processing system in ACL2s. For future work, we plan to precisely classify temporal logic properties that are preserved by skipping refinement. This would enable us to transfer temporal properties from specifications to implementations, after establishing refinement.

References

1. Aagaard, M.D., Cook, B., Day, N.A., Jones, R.B.: A framework for microprocessor correctness statements. In: Margaria, T., Melham, T. (eds.) CHARME 2001. LNCS, vol. 2144, pp. 433–448. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44798-9_33
2. Basten, T.: Branching bisimilarity is an equivalence indeed!. Inf. Process. Lett. **58**, 141–147 (1996)
3. Chamarthi, H.R., Dillinger, P., Manolios, P., Vroon, D.: The ACL2 sedan theorem proving system. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 291–295. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19835-9_27
4. Clarke, E.M., Grumberg, O., Browne, M.C.: Reasoning about networks with many identical finite-state processes. In: PODC (1986)
5. van Glabbeek, R.J.: The linear time-branching time spectrum (extended abstract). In: Baeten, J.C.M., Klop, J.W. (eds.) CONCUR 1990. LNCS, vol. 458, pp. 278–297. Springer, Heidelberg (1990). <https://doi.org/10.1007/BFb0039066>

6. Gu, L., Vaynberg, A., Ford, B., Shao, Z., Costanzo, D.: CertiKOS: a certified kernel for secure cloud computing. In: APSys (2011)
7. Gu, R., et al.: Deep specifications and certified abstraction layers. In: POPL (2015)
8. Hawblitzel, C., et al.: IronFleet: Proving practical distributed systems correct. In: SOSP (2015)
9. Hosabettu, R., Gopalakrishnan, G., Srivastava, M.: Formal verification of a complex pipelined processor. *Form. Methods Syst. Des.* **23**, 171–213 (2003)
10. Jain, M., Manolios, P.: Skipping refinement. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 103–119. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_7
11. Klarlund, N.: Progress measures and finite arguments for infinite computations. Ph.D. thesis (1990)
12. Klein, G., Sewell, T., Winwood, S.: Refinement in the formal verification of the seL4 microkernel. In: Hardin, D. (ed.) Design and Verification of Microprocessor Systems for High-Assurance Applications, pp. 323–339. Springer, Boston (2010). https://doi.org/10.1007/978-1-4419-1539-9_11
13. Kragl, B., Qadeer, S.: Layered concurrent programs. In: Chockler, H., Weissbucher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 79–102. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_5
14. Lamport, L.: What good is temporal logic. *Information processing* (1993)
15. Leroy, X., Blazy, S.: Formal verification of a c-like memory model and its uses for verifying program transformations. *J. Autom. Reason.* **41**, 1–31 (2008)
16. Liu, X., Yu, T., Zhang, W.: Analyzing divergence in bisimulation semantics. In: POPL (2017)
17. Lynch, N.A., Vaandrager, F.W.: Forward and backward simulations: I. Untimed systems. *Inf. Comput.* (1995)
18. Manolios, P.: Mechanical verification of reactive systems. Ph.D. thesis, University of Texas (2001)
19. Manolios, P.: Correctness of pipelined machines. In: Hunt, W.A., Johnson, S.D. (eds.) FMCAD 2000. LNCS, vol. 1954, pp. 181–198. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-40922-X_11
20. Manolios, P.: A compositional theory of refinement for branching time. In: Geist, D., Tronci, E. (eds.) CHARME 2003. LNCS, vol. 2860, pp. 304–318. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-39724-3_28
21. Manolios, P., Srinivasan, S.K.: A complete compositional reasoning framework for the efficient verification of pipelined machines. In: ICCAD (2005)
22. Milner, R.: An algebraic definition of simulation between programs. In: Proceedings of the 2nd International Joint Conference on Artificial Intelligence (1971)
23. Misra, J.: Distributed discrete-event simulation. *ACM Comput. Surv.* **18**, 39–65 (1986)
24. Namjoshi, K.S.: A simple characterization of stuttering bisimulation. In: Ramesh, S., Sivakumar, G. (eds.) FSTTCS 1997. LNCS, vol. 1346, pp. 284–296. Springer, Heidelberg (1997). <https://doi.org/10.1007/BFb0058037>
25. Namjoshi, K.S., Zuck, L.D.: Witnessing program transformations. In: Logozzo, F., Fähndrich, M. (eds.) SAS 2013. LNCS, vol. 7935, pp. 304–323. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38856-9_17
26. Ray, S., Jr Hunt, W.A.: Deductive verification of pipelined machines using first-order quantification. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 31–43. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27813-9_3

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Robust Controller Synthesis in Timed Büchi Automata: A Symbolic Approach

Damien Busatto-Gaston^{1(✉)},
Benjamin Monmege¹, Pierre-Alain Reynier¹,
and Ocan Sankur²



¹ Aix Marseille Univ, Université de Toulon,
CNRS, LIS, Marseille, France

{damien.busatto,pierre-alain.reynier}@lis-lab.fr,
benjamin.monmege@univ-amu.fr

² Univ Rennes, Inria, CNRS, IRISA, Rennes, France
ocan.sankur@irisa.fr

Abstract. We solve in a purely symbolic way the robust controller synthesis problem in timed automata with Büchi acceptance conditions. The goal of the controller is to play according to an accepting lasso of the automaton, while resisting to timing perturbations chosen by a competing environment. The problem was previously shown to be PSPACE-complete using regions-based techniques, but we provide a first tool solving the problem using zones only, thus more resilient to state-space explosion problem. The key ingredient is the introduction of branching constraint graphs allowing to decide in polynomial time whether a given lasso is robust, and even compute the largest admissible perturbation if it is. We also make an original use of constraint graphs in this context in order to test the inclusion of timed reachability relations, crucial for the termination criterion of our algorithm. Our techniques are illustrated using a case study on the regulation of a train network.

1 Introduction

Timed automata [1] extend finite-state automata with timing constraints, providing an automata-theoretic framework to design, model, verify and synthesise real-time systems. However, the semantics of timed automata is a mathematical idealisation: it assumes that clocks have infinite precision and instantaneous actions. Proving that a timed automaton satisfies a property does not ensure that a real implementation of it also does. This *robustness* issue is a challenging problem for embedded systems [12], and alternative semantics have been proposed, so as to ensure that the verified (or synthesised) behaviour remains correct in presence of small timing perturbations.

We are interested in a fundamental controller synthesis problem in timed automata equipped with a Büchi acceptance condition: it consists in determining whether there exists an accepting infinite execution.

This work was funded by ANR project Ticktac (ANR-18-CE40-0015).

Thus, the role of the controller is to choose transitions and delays. This problem has been studied numerously in the exact setting [13–15, 17, 19, 27, 28]. In the context of robustness, this strategy should be tolerant to small perturbations of the delays. This discards strategies suffering from weaknesses such as Zeno behaviours, or even non-Zeno behaviours requiring infinite precision, as exhibited in [6].

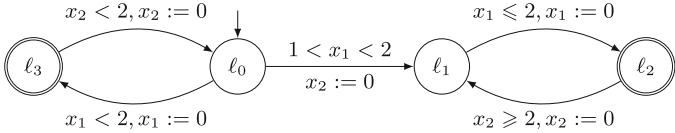
More formally, the semantics we consider is defined as a game that depends on some parameter δ representing an upper bound on the amplitude of the perturbation [7]. In this game, the controller plays against an antagonistic environment that can perturb each delay using a value chosen in the interval $[-\delta, \delta]$. The case of a fixed value of δ has been shown to be decidable in [7], and also for a related model in [18]. However, these algorithms are based on regions, and as the value of δ may be very different from the constants appearing in the guards of the automaton, do not yield practical algorithms. Moreover, the maximal perturbation is not necessarily known in advance, and could be considered as part of the design process.

The problem we are interested in is *qualitative*: we want to determine whether *there exists* a positive value of δ such that the controller wins the game. It has been proven in [25] that this problem is in PSPACE (and even PSPACE-complete), thus no harder than in the exact setting with no perturbation allowed [1]. However, the algorithm heavily relies on regions, and more precisely on an abstraction that refines the one of regions, namely folded orbit graphs. Hence, it is not at all amenable to implementation.

Our objective is to provide an efficient symbolic algorithm for solving this problem. To this end, we target the use of *zones* instead of regions, as they allow an on-demand partitioning of the state space. Moreover, the algorithm we develop explores the reachable state-space in a *forward* manner. This is known to lead to better performances, as witnessed by the successful tool UPPAAL TIGA based on forward algorithms for solving controller synthesis problems [5].

Our algorithm can be understood as an adaptation to the robustness setting of the standard algorithm for Büchi acceptance in timed automata [17]. This algorithm looks for an accepting lasso using a double depth-first search. A major difficulty consists in checking whether a lasso can be robustly iterated, i.e. whether there exists $\delta > 0$ such that the controller can follow the cycle for an infinite amount of steps while being tolerant to perturbations of amplitude at most δ . The key argument of [25] was the notion of aperiodic folded orbit graph of a path in the region automaton, thus tightly connected to regions. Lifting this notion to zones seems impossible as it makes an important use of the fact that valuations in regions are time-abstract bisimilar, which is not the case for zones.

Our contributions are threefold. First, we provide a polynomial time procedure to decide, given a lasso, whether it can be robustly iterated. This symbolic algorithm relies on a computation of the greatest fixpoint of the operator describing the set of controllable predecessors of a path. In order to provide an argument of termination for this computation, we resort to a new notion of branching constraint graphs, extending the approach used in [16, 26] and based

**Fig. 1.** A timed automaton

on constraint graphs (introduced in [8]) to check iterability of a cycle, without robustness requirements. Second, we show that when considering a lasso, not only can we decide robust iterability, but we can even compute the largest perturbation under which it is controllable. This problem was not known to be decidable before. Finally, we provide a termination criterion for the analysis of lassos. Focusing on zones is not complete: it can be the case that two cycles lead to the same zones, but one is robustly iterable while the other one is not. Robust iterability crucially depends on the real-time dynamics of the cycle and we prove that it actually only depends on the reachability relation of the path. We provide a polynomial-time algorithm for checking inclusion between reachability relations of paths in timed automata based on constraint graphs. It is worth noticing that all our procedures can be implemented using difference bound matrices, a very efficient data structure used for timed systems. These developments have been integrated in a tool, and we present a case study of a train regulation network illustrating its performances.

Integrating the robustness question in the verification of real-time systems has attracted attention in the community, and the recent works include, for instance, robust model checking for timed automata under clock drifts [23], Lipschitz robustness notions for timed systems [11], quantitative robust synthesis for timed automata [2]. Stability analysis and synthesis of stabilizing controllers in hybrid systems are a closely related topic, see e.g. [20, 21].

2 Timed Automata: Reachability and Robustness

Let $\mathcal{X} = \{x_1, \dots, x_n\}$ be a finite set of clock variables. It is extended with a virtual clock x_0 , constantly equal to 0, and we denote by \mathcal{X}_0 the set $\mathcal{X} \cup \{x_0\}$. An atomic clock constraint on \mathcal{X} is a formula $x - y \leq k$, or $x - y < k$ with $x \neq y \in \mathcal{X}_0$ and $k \in \mathbb{Q}$. A constraint is non-diagonal if one of the two clocks is x_0 . We denote by $\text{Guards}(X)$ (respectively, $\text{Guards}_{\text{nd}}(X)$) the set of (clock) constraints (respectively, non-diagonal clock constraints) built as conjunctions of atomic clock constraints (respectively, non-diagonal atomic clock constraints).

A clock valuation ν is an element of $\mathbb{R}_{\geq 0}^{\mathcal{X}}$. It is extended to $\mathbb{R}_{\geq 0}^{\mathcal{X}_0}$ by letting $\nu(x_0) = 0$. For all $d \in \mathbb{R}_{>0}$, we let $\nu + d$ be the valuation defined by $(\nu + d)(x) = \nu(x) + d$ for all clocks $x \in \mathcal{X}$. If $\mathcal{Y} \subseteq \mathcal{X}$, we also let $\nu[\mathcal{Y} \leftarrow 0]$ be the valuation resetting clocks in \mathcal{Y} to 0, without modifying values of other clocks. A valuation ν satisfies an atomic clock constraint $x - y \bowtie k$ (with $\bowtie \in \{\leq, <\}$) if $\nu(x) - \nu(y) \bowtie k$. The satisfaction relation is then extended to clock constraints

naturally: the satisfaction of constraint g by a valuation ν is denoted by $\nu \models g$. The set of valuations satisfying a constraint g is denoted by $[\![g]\!]$.

A *timed automaton* is a tuple $\mathcal{A} = (L, \ell_0, E, L_t)$ with L a finite set of locations, $\ell_0 \in L$ an initial location, $E \subseteq L \times \text{Guards}_{\text{nd}}(\mathcal{X}) \times 2^{\mathcal{X}} \times L$ is a finite set of edges, and L_t is a set of accepting locations.

An example of timed automaton is depicted in Fig. 1, where the reset of a clock x is denoted by $x := 0$. The semantics of the timed automaton \mathcal{A} is defined as an infinite transition system $[\![\mathcal{A}]\!] = (S, s_0, \rightarrow)$. The set S of states of $[\![\mathcal{A}]\!]$ is $L \times \mathbb{R}_{\geq 0}^{\mathcal{X}}$, $s_0 = (\ell_0, \mathbf{0})$. A transition of $[\![\mathcal{A}]\!]$ is of the form $(\ell, \nu) \xrightarrow{e, d} (\ell', \nu')$ with $e = (\ell, g, \mathcal{Y}, \ell') \in E$ and $d \in \mathbb{R}_{>0}$ such that $\nu + d \models g$ and $\nu' = (\nu + d)[\mathcal{Y} \leftarrow 0]$. We call *path* a possible finite sequence of edges in the timed automaton. The *reachability relation* of a path ρ , denoted by $\text{Reach}(\rho)$ is the set of pairs (ν, ν') such that there is a sequence of transitions of $[\![\mathcal{A}]\!]$ starting from (ℓ, ν) , ending in (ℓ', ν') and that follows ρ in order as the edges of the timed automaton. A *run* of \mathcal{A} is an infinite sequence of transitions of $[\![\mathcal{A}]\!]$ starting from s_0 . We are interested in Büchi objectives. Therefore, a run is accepting if there exists a final location $\ell_t \in L_t$ that the run visits infinitely often.

As done classically, we assume that every clock is bounded in \mathcal{A} by a constant M , that is we only consider the previous infinite transition system over the subset $L \times [0, M]^{\mathcal{X}}$ of states.

We study the robustness problem introduced in [25], that is stated in terms of games where a controller fights against an environment. After a prefix of a run, the controller will have the capability to choose delays and transitions to fire, whereas the environment perturbs the delays chosen by the controller with a small parameter $\delta > 0$. The aim of the controller will be to find a strategy so that, no matter how the environment plays, he is ensured to generate an infinite run satisfying the Büchi condition. Formally, given a timed automaton $\mathcal{A} = (L, \ell_0, E, L_t)$ and $\delta > 0$, the perturbation game is a two-player turn-based game $\mathcal{G}_\delta(\mathcal{A})$ between a controller and an environment. Its state space is partitioned into $S_C \sqcup S_E$ where $S_C = L \times \mathbb{R}_{\geq 0}^{\mathcal{X}}$ belongs to the controller, and $S_E = L \times \mathbb{R}_{\geq 0}^{\mathcal{X}} \times \mathbb{R}_{>0} \times E$ to the environment. The initial state is $(\ell_0, \mathbf{0}) \in S_C$. From each state $(\ell, \nu) \in S_C$, there is a transition to $(\ell, \nu, d, e) \in S_E$ with $e = (\ell, g, \mathcal{Y}, \ell') \in E$ whenever $d > \delta$, and $\nu + d + \varepsilon \models g$ for all $\varepsilon \in [-\delta, \delta]$. Then, from each state $(\ell, \nu, d, (\ell, g, \mathcal{Y}, \ell')) \in S_E$, there is a transition to $(\ell', (\nu + d + \varepsilon)[r \leftarrow 0]) \in S_C$ for all $\varepsilon \in [-\delta, \delta]$. A play of $\mathcal{G}_\delta(\mathcal{A})$ is a finite or infinite path $q_0 \xrightarrow{t_1} q_1 \xrightarrow{t_2} q_2 \dots$ where $q_0 = (\ell_0, 0)$ and t_i is a transition from state q_{i-1} to q_i , for all $i > 0$. It is said to be maximal if it is infinite or can not be extended with any transition.

A strategy for the controller is a function σ_{Con} mapping each non-maximal play ending in some $(\ell, \nu) \in S_C$ to a pair (d, e) where $d > 0$ and $e \in E$ such that there is a transition from (ℓ, ν) to (ℓ, ν, d, e) . A strategy for the environment is a function σ_{Env} mapping each finite play ending in (ℓ, ν, d, e) to a state (ℓ', ν') related by a transition. A play gives rise to a unique run of $[\![\mathcal{A}]\!]$ by only keeping states in V_C . For a pair of strategies $(\sigma_{\text{Con}}, \sigma_{\text{Env}})$, we let $\text{play}_\mathcal{A}^\delta(\sigma_{\text{Con}}, \sigma_{\text{Env}})$ denote the run associated with the unique maximal play of $\mathcal{G}_\delta(\mathcal{A})$ that follows the strategies. Controller's strategy σ_{Con} is winning (with respect to the Büchi

objective L_t) if for all strategies σ_{Env} of the environment, $\text{play}_{\mathcal{A}}^{\delta}(\sigma_{\text{Con}}, \sigma_{\text{Env}})$ is infinite and visits infinitely often some location of L_t . The *parametrised robust controller synthesis problem* asks, given a timed automaton \mathcal{A} , whether there exists $\delta > 0$ such that the controller has a winning strategy in $\mathcal{G}_{\delta}(\mathcal{A})$.

Example 1. The controller has a winning strategy in $\mathcal{G}_{\delta}(\mathcal{A})$, with \mathcal{A} the automaton of Fig. 1, for all possible values of $\delta < 1/2$. Indeed, he can follow the cycle $\ell_0 \rightarrow \ell_3 \rightarrow \ell_0$ by always picking time delay $1/2$ so that, when arriving in ℓ_3 (resp. ℓ_0) after the perturbation of the environment, clock x_2 (resp. x_1) has a valuation in $[1/2 - \delta, 1/2 + \delta]$. Therefore, he can play forever following this memoryless strategy. For $\delta \geq 1/2$, the environment can enforce reaching ℓ_3 with a value for x_2 at least equal to 1. The guard $x_2 < 2$ of the next transition to ℓ_0 cannot be guaranteed, and therefore the controller cannot win $\mathcal{G}_{\delta}(\mathcal{A})$. In [25], it is shown that the cycle around ℓ_2 does not provide a winning strategy for the controller for any value of $\delta > 0$ since perturbations accumulate so that the controller can only play it a finite number of times in the worst case.

By [25], the parametrised robust controller synthesis problem is known to be PSPACE-complete. Their solution is based on the region automaton of \mathcal{A} . We are seeking for a more practical solution using zones. A zone Z over \mathcal{X} is a convex subset of $\mathbb{R}_{\geq 0}^{\mathcal{X}}$ defined as the set of valuations satisfying a clock constraint g , i.e. $Z = \llbracket g \rrbracket$. Zones can be encoded into *difference-bound matrices* (DBM), that are $|\mathcal{X}_0| \times |\mathcal{X}_0|$ -matrices over $(\mathbb{R} \times \{<, \leq\}) \cup \{(\infty, <)\}$. We adopt the following notation: for a DBM M , we write $M = (\mathbf{M}, \prec^M)$, where \mathbf{M} is the matrix made of the first components, with elements in $\mathbb{R} \cup \{\infty\}$, while \prec^M is the matrix of the second components, with elements in $\{<, \leq\}$. A DBM M naturally represents a zone (which we abusively write M as well), defined as the set of valuations ν such that, for all $x, y \in \mathcal{X}_0$, $\nu(x) - \nu(y) \prec^M_{x,y} M_{x,y}$ (where $\nu(x_0) = 0$). Coefficients of a DBM are thus pairs (\prec, c) . As usual, these can be compared: (\prec, c) is less than (\prec', c') (denoted by $(\prec, c) < (\prec', c')$) whenever $c < c'$ or $(c = c', \prec = < \text{ and } \prec' = \leq)$. Moreover, these coefficients can be added: $(\prec, c) + (\prec', c')$ is the pair $(\prec'', c + c')$ with $\prec'' = \leq$ if $\prec = \prec' = \leq$ and $\prec'' = <$ otherwise.

DBMs were introduced in [4, 10] for analyzing timed automata; we refer to [3] for details. Standard operations used to explore the state space of timed automata have been defined on DBMs: intersection is written $M \cap N$, $\text{Pretime}_{>t}(M)$ is the set of valuations such that a time delay of more than t time units leads to the zone M , $\text{Unreset}_R(M)$ is the set of valuations that end in M when the clocks in R are reset. From a robustness perspective, we also consider the operator $\text{shrink}_{[-\delta, \delta]}(M)$ defined as the set of valuations ν such that $\nu + [-\delta, \delta] \subseteq M$ introduced in [24]. Given a DBM M and a rational number δ , all these operations can be effectively computed in time cubic in $|\mathcal{X}|$.

3 Reachability Relation of a Path

Before treating the robustness issues, we start by designing a symbolic (i.e. zone-based) approach to describe and compare the reachability relations of paths

in timed automata. This will be crucial subsequently to design a termination criterion in the state space exploration of our robustness-checking algorithm. Solving the inclusion of reachability relations in a symbolic manner has independent interest and can have other applications.

The reachability relation $\text{Reach}(\rho)$ of a path ρ , is a subset of $\mathbb{R}_{\geq 0}^{\mathcal{X} \cup \mathcal{X}'}$ where \mathcal{X}' are primed versions of the clocks, such that each $(\nu, \nu') \in \text{Reach}(\rho)$ iff there is a run from valuation ν to valuation ν' following ρ . Unfortunately, reachability relations $\text{Reach}(\rho)$ are not zones in general, that is, they cannot be represented using only difference constraints. In fact, we shall see shortly that constraints of the form $x - y + z - u \leq c$ also appear, as already observed in [22]. We thus cannot rely directly on the traditional difference bound matrices (DBMs) used to represent zones. We instead rely on the constraint graphs that were introduced in [8], and explored in [16] for the parametric case (the latter work considers enlarged constraints, and not shrunk ones as we study here). Our contribution is to use these graphs to obtain a syntactic check of inclusion of the according reachability relations.

Constraint Graphs. Rather than considering the values of the clocks in \mathcal{X} , this data structure considers the date X_i of the latest reset of the clock x_i , and uses a new variable τ denoting the global timestamp. Note that the clock values can be recovered easily since $X_i = \tau - x_i$. For the extra clock x_0 , we introduce variable X_0 equal to the global timestamp τ (since x_0 must remain equal to 0). A constraint graph defining a zone is a weighted graph whose nodes are $X = \{X_0, X_1, \dots, X_n\}$. Constraints on clocks are represented by weights on edges in the graph: a constraint $X - Y \prec c$ is represented by an edge from X to Y weighted by (\prec, c) , with $\prec \in \{\leq, <\}$ and $c \in \mathbf{Q}$. Weights in the graph are thus pairs of the form (\prec, c) . Therefore, we can compute shortest weights between two vertices of a weighted graph. A cycle is said to be negative if it has weight at most $(<, 0)$, i.e. $(<, 0)$ or (\prec, c) with $c < 0$.

Encoding Paths. Constraint graphs can also encode tuples of valuations seen along a path. To encode a k -step computation, we make $k + 1$ copies of the nodes, that is, $X^i = \{X_0^i, X_1^i, \dots, X_n^i\}$ for $i \in \{1, \dots, k + 1\}$. These copies are also called *layers*. Let us first consider an example on the path ρ consisting of the edge from ℓ_1 to ℓ_2 , and the edge from ℓ_2 to ℓ_1 , in the timed automaton of Fig. 1. The constraint graph G_ρ is depicted in Fig. 3: in our diagrams of constraint graphs, the absence of labels on an edge means $(\leq, 0)$, and we depict with an edge with arrows on both ends the presence of an edge in both directions. The graph has five columns, each containing copies of the variables for that step: they represent the valuations before the first edge, after the first time elapse, after the first reset, after the second time elapse and after the second reset. In general now, each elementary operation can be described by a constraint graph with two layers (X_i) (before) and (X'_i) (after).

- The operation $\text{Pftime}_{>t}$ is described by the constraint graph $G_{\text{time}}^{>t}$ with edges $X_i \rightarrow X_0$, $X_i \leftrightarrow X'_i$ for $i > 0$, and $X_0 \xrightarrow{(<, -t)} X'_0$. Figure 3 contains two occurrences of $G_{\text{time}}^{>0}$: we always represent with dashed arrows edges that are

labelled by $(<, c)$, and plain arrows edges that are labelled with (\leqslant, c) ; the absence of an edge means that it is labelled with $(<, \infty)$.

- The operation $g \cap \text{Unreset}_Y(\cdot)$, to test a guard g and reset the clocks in \mathcal{Y} , is described by the constraint graph $G_{\text{edge}}^{g, \mathcal{Y}}$ with edges $X_0 \leftrightarrow X'_0$ (meaning that the time does not elapse), $X_i \leftrightarrow X'_i$ for i such that clock $x_i \notin \mathcal{Y}$, and $X'_i \leftrightarrow X'_0$ for i such that clock $x_i \in \mathcal{Y}$, and for all clock constraint $x_i - x_j \prec c$ appearing in g , an edge from X_j to X_i labelled by (\prec, c) (since it encodes the fact that $(\tau - X_i) - (\tau - X_j) = X_j - X_i \prec c$). In Fig. 3, we have first $G_{\text{edge}}^{x_1 \leqslant 2, \{x_1\}}$, and then $G_{\text{edge}}^{x_2 \geqslant 2, \{x_2\}}$.

Constraint graphs can be stacked one after the other to obtain the constraint graph of an edge e , and then of a path ρ , that we denote by G_ρ . In the resulting graph, there is one leftmost layer of vertices $(X_i^\ell)_i$ and one rightmost one $(X_i^r)_i$ representing the situation before and after the firing of the path ρ . Once this graph is constructed, the intermediary levels can be eliminated after replacing each edge between the nodes of $X^\ell \cup X^r$ by the shortest path in the graph. This phase is hereafter called *normalisation* of the constraint graph. The normalised version of the constraint graph of Fig. 3 is depicted on its right.

From Constraint Graphs to Reachability Relations. From a logical point of view, the elimination of intermediary layers reflects an elimination of quantifiers in a formula of the first-order theory of real numbers. At the end, we obtain a set of constraints of the form $X_i^k - X_j^{k'} \prec c$ with $k, k' \in \{\ell, r\}$. These constraints do not reflect uniquely the reachability relation $\text{Reach}(\rho)$, in the sense that it is possible that $\text{Reach}(\rho_1) = \text{Reach}(\rho_2)$ but the normalised versions of G_{ρ_1} and G_{ρ_2} are different. For example, if we consider the path ρ^2 obtained by repeating the cycle ρ between ℓ_1 and ℓ_2 , the reachability relation does not change ($\text{Reach}(\rho^2) = \text{Reach}(\rho)$), but the normalised constraint graph does ($G_{\rho^2} \neq G_\rho$): all labels $(\leqslant, 2)$ of the red dotted edges from the rightmost layer to the leftmost layer become $(\leqslant, 4)$, and the labels $(\leqslant, -2)$ of the dashed blue edges become $(\leqslant, -4)$.

We solve this issue by jumping back from variables X_i^k to the clock valuations. Indeed, in terms of clock valuations ν^ℓ and ν^r before and after the path, the constraint $X_i^k - X_j^{k'} \prec c$ (for $k, k' \in \{\ell, r\}$) rewrites as $(\tau^k - \nu^k(x_i)) - (\tau^{k'} - \nu^{k'}(x_j)) \prec c$, where τ^ℓ is the global timestamp before firing ρ and τ^r the one after. When $k = k'$, variables τ^ℓ and τ^r disappear, leaving a constraint of the form $\nu^k(x_j) - \nu^k(x_i) \prec c$. When $k \neq k'$, we can rewrite the constraint as $\tau^k - \tau^{k'} \prec \nu^k(x_i) - \nu^{k'}(x_j) + c$. We therefore obtain upper and lower bounds on the value of $\tau^r - \tau^\ell$, allowing us to eliminate $\tau^r - \tau^\ell$ considered as a single variable. We therefore obtain in fine a formula mixing constraints of the form

- $\nu^k(x_a) - \nu^k(x_b) \prec p$, with $k \in \{\ell, r\}$, $a \neq b$, and we define $\gamma_{a,b}^k = (\prec, p)$;
- $\nu^\ell(x_a) - \nu^\ell(x_b) + \nu^r(x_c) - \nu^r(x_d) \prec p$, with $a \neq b$ and $c \neq d$, and we define $\gamma_{a,b,c,d} = (\prec, p)$. This constraint can appear in two ways: either from $\nu^r(x_c) - \nu^\ell(x_b) + p_1 \prec_1 \tau^r - \tau^\ell \prec_2 \nu^\ell(x_a) - \nu^r(x_d) + p_2$ by eliminating $\tau^r - \tau^\ell$, or by adding the two constraints of the form $\nu^\ell(x_a) - \nu^\ell(x_b) \prec_1 p_1$ and

$\nu^r(x_c) - \nu^r(x_d) \prec_2 p_2$. Thus, $\gamma_{a,b,c,d}$ is obtained as the minimum of the two constraints obtained in this manner. In other terms, in the constraint graph, this constraint is the minimal weight between the sum of the weights of the edges (X_d^r, X_a^l) and (X_b^l, X_c^r) , and the sum of the weights of the edges (X_b^l, X_a^l) and (X_d^r, X_c^r) . For example, in the path in Fig. 3, we have $\gamma_{0,1,0,2} = (\leq, 0)$ since the two constraints are $(\leq, 0)$ and $(<, \infty)$, whereas $\gamma_{1,2,2,1} = (\leq, 0)$ because the two constraints are $(<, 2)$ and $(\leq, 0)$.

Let $\varphi(G)$ be the conjunction of such constraints obtained from a constraint graph G once normalised: this is a quantifier-free formula of the additive theory of reals. We obtain the following property whose proof mimics the one for proving the normalisation of DBMs (and can be derived from the developments of [8]).

Lemma 1. *Let ρ be a path in a timed automaton. If G_ρ contains a negative cycle, then $\text{Reach}(\rho) = \emptyset$. Otherwise, $\text{Reach}(\rho)$ is the set of pairs of valuations (ν^ℓ, ν^r) that satisfy the formula $\varphi(G_\rho)$.*

Checking Inclusion. For a path ρ , we regroup the pairs $(\gamma_{a,b}^l)$, $(\gamma_{a,b}^r)$ and $(\gamma_{a,b,c,d})$ above in a single vector Γ^ρ . We extend the comparison relation $<$ to these vectors by applying it componentwise. These vectors can be used to check equality or inclusion of reachability relations in time $O(|X|^4)$:

Theorem 1. *Let ρ and ρ' be paths in a timed automaton such that $\text{Reach}(\rho)$ and $\text{Reach}(\rho')$ are non empty. Then $\text{Reach}(\rho) \subseteq \text{Reach}(\rho')$ if and only if $\Gamma^\rho \leqslant \Gamma^{\rho'}$.*

Notice that we do not need to check equivalence or implication of formulas $\varphi(G_\rho)$ and $\varphi(G_{\rho'})$, but simply check syntactically constants appearing in these formulas. Moreover, these constants can be stored in usual DBMs on $2 \times |\mathcal{X}_0|$ clocks, allowing for reusability of classical DBM libraries. For the constraint graph in Fig. 3, we have seen that $G_{\rho^2} \neq G_{\rho^1}$, even if $\text{Reach}(\rho^2) = \text{Reach}(\rho)$. However, we can check that $\varphi(G_{\rho^2}) = \varphi(G_\rho)$ as expected.

Computation of Pre and Post. By Lemma 1 and the construction of constraint graphs, one can easily compute $\text{Pre}_\rho(Z) = \{\nu \mid \exists \nu' \in Z ((\ell, \nu), (\ell', \nu')) \in \text{Reach}(\rho)\}$ for a given path ρ and zone Z (see [8, 16]). In fact, consider the normalised constraint graph G_ρ on nodes $X^\ell \cup X^r$. To compute $\text{Pre}_\rho(Z)$, one just needs to add the constraints of Z on X^r . This is done by replacing each edge $X_i^r \xrightarrow{w} X_j^r$ by $X_i^r \xrightarrow{\min(Z_{j,i}, w)} X_j^r$ where $Z_{j,i} = (\prec, p)$ defines the constraint of Z on $x_j - x_i$. Then, the normalisation of the graph describes the reachability relation along path ρ ending in zone Z . Furthermore, projecting the constraints to X^ℓ yields $\text{Pre}_\rho(Z)$: this can be obtained by gathering all constraints on pairs of nodes of X^ℓ . A reachability relation can thus be seen as a function assigning to each zone Z its image by ρ . One can symmetrically compute the successor $\text{Post}_\rho(Z) = \{\nu' \mid \exists \nu \in Z ((\ell, \nu), (\ell', \nu')) \in \text{Reach}(\rho)\}$ by constraining the nodes X^ℓ and projecting to X^r .

4 Robust Iterability of a Lasso

In this section, we study the perturbation game $\mathcal{G}_\delta(\mathcal{A})$ between the two players (controller and environment), as defined in Sect. 2, when the timed automaton \mathcal{A} is restricted to a fixed *lasso* $\rho_1\rho_2$, i.e. ρ_1 is a path from ℓ_0 to some accepting location ℓ_t , and ρ_2 a cyclic path around ℓ_t . This implies that the controller does not have the choice of the transitions, but only of the delays. We will consider different settings, in which δ is fixed or not.

Controllable Predecessors and their Greatest Fixpoints. Consider an edge $e = (\ell, g, R, \ell')$. For any set $Z \subseteq \mathbb{R}_{\geq 0}^{\mathcal{X}}$, we define the *controllable predecessors of Z* as follows: $\text{CPre}_e^\delta(Z) = \text{Pretime}_{>\delta}(\text{shrink}_{[-\delta, \delta]}(g \cap \text{Unreset}_R(Z)))$. Intuitively, $\text{CPre}_e^\delta(Z)$ is the set of valuations from which the controller can ensure reaching Z in one step, following the edge e , no matter of the perturbations of amplitude at most δ of the environment. In fact, it can delay in $\text{shrink}_{[-\delta, \delta]}(g \cap \text{Unreset}_R(Z))$ with a delay of at least δ , where under any perturbation in $[-\delta, \delta]$, the valuation satisfies the guard, and it ends, after reset, in Z . Results of [24] show that this operator can be computed in cubic time with respect to the number of clocks. We extend this operator to a path ρ by composition, denoted it by CPre_ρ^δ . Note that $\text{CPre}_\rho^0 = \text{Pre}_\rho$ is the usual predecessor operator without perturbation.

This operator is monotone, hence its greatest fixpoint $\nu X \text{CPre}_\rho^\delta(X)$ is well-defined, equal to $\bigcap_{i \geq 0} \text{CPre}_{\rho^i}^\delta(\top)$: it corresponds to the valuations from which the controller can guarantee to loop forever along the path ρ . By definition of the game $\mathcal{G}_\delta(\mathcal{A})$ where \mathcal{A} is restricted to the lasso $\rho_1\rho_2$, the controller wins the game if and only if $\mathbf{0} \in \text{CPre}_{\rho_1}^\delta(\nu X \text{CPre}_{\rho_2}^\delta(X))$. As a consequence, our problem reduces to the computation of this greatest fixpoint.

Branching Constraint Graphs. We consider first a fixed (rational) value of the parameter δ , and are interested in the computation of the greatest fixpoint $\nu X \text{CPre}_{\rho_2}^\delta(X)$. In [16], constraints graphs were used to provide a termination criterion allowing to compute the greatest fixpoint of the classical predecessor operator CPre_ρ^0 . We generalize this approach to deal with the operator CPre_ρ^δ and to this end, we need to generalize constraint graphs so as to encode it. Unfortunately, the operator $\text{shrink}_{[-\delta, \delta]}$ cannot be encoded in a constraint graph. Intuitively, this comes from the fact that a constraint graph represents a relation between valuations, while there is no such relation associated with the CPre_ρ^δ operator. Instead, we introduce *branching constraint graphs*, that will faithfully represent the CPre_ρ^δ operator: unlike constraint graphs introduced so far that have a left layer and a right layer of variables, a branching constraint graph has still a single left layer but several right layers.

We first define a branching constraint graph G_{shrink}^δ associated with the operator $\text{shrink}_{[-\delta, \delta]}$ as follows. Its set of vertices is composed of three copies of the $\{X_0, X_1, \dots, X_n\}$, denoted by primed, unprimed and doubly primed versions. Edges are defined so as to encode the following constraints : $X'_i = X_i$ and $X''_i = X_i$ for every $i \neq 0$, and $X'_0 = X_0 + \delta$ and $X''_0 = X_0 - \delta$. An instance of this graph can be found in several occurrences in Fig. 2.

Proposition 1. Let Z be a zone and $G_{\text{shrink}}^\delta(Z)$ be the graph obtained from G_{shrink}^δ by adding on primed and doubly primed vertices the constraints defining Z (as for $\text{Pre}_\rho(Z)$ in the end of Sect. 3). Then the constraint on unprimed vertices obtained from the shortest paths in $G_{\text{shrink}}^\delta(Z)$ is equivalent to $\text{shrink}_{[-\delta, \delta]}(Z)$.

Proof. Given a zone Z and a real number d , we define $Z + d = \{\nu + d \mid \nu \in Z\}$. One easily observes that $\text{shrink}_{[-\delta, \delta]}(Z) = (Z + \delta) \cap (Z - \delta)$. The result follows from the observation that taking two distinct copies of vertices, and considering shortest paths allows one to encode the intersection. \square

Then, for all edges $e = (\ell, g, R, \ell')$, we define the branching constraint graph G_e^δ as the graph obtained by stacking (in this order) the branching constraint graph $G_{\text{time}}^{>0}$, G_{shrink}^δ and $G_{\text{edge}}^{g, \mathcal{Y}}$. Note that two copies of the graph $G_{\text{edge}}^{g, \mathcal{Y}}$ are needed, to be connected to the two sets of vertices that are on the right of the graph G_{shrink}^δ . This definition is extended in the expected way to a finite path ρ , yielding the graph G_ρ^δ . In this graph, there is a single set of vertices on the left, and $2^{|\rho|}$ sets of vertices on the right. As a direct consequence of the previous results on the constraint graphs for time elapse, shrinking and guard/reset, one obtains:

Proposition 2. Let Z be a zone and ρ be a path. We let $G_\rho^\delta(Z)$ be the graph obtained from G_ρ^δ by adding on every set of right vertices the constraints defining Z . Then the constraint on the left layer of vertices obtained from the shortest paths in $G_\rho^\delta(Z)$ is equivalent to $\text{CPre}_\rho^\delta(Z)$.

An example of the graph $G_\rho^\delta(Z)$ for $\rho = e_1 e_2$, edges considered in Fig. 3, is depicted in Fig. 2 (on the left).

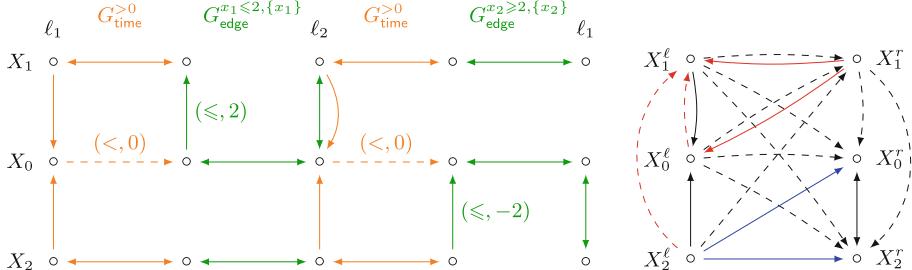


Fig. 2. On the left, the branching constraint graph $G_{e_1 e_2}^\delta$ encoding the operator $\text{CPre}_{e_1 e_2}^\delta$, where e_1 and e_2 refer to edges considered in Fig. 3. Dashed edges have weight $(<, .)$, plain edges have weight $(\leq, .)$. Black edges (resp. orange edges, pink edges, red edges, blue edges) are labelled by $(., 0)$ (resp. $(., -\delta)$, $(., \delta)$, $(., 2)$, $(., -2)$). On the right, a decomposition of a path in a branching constraint graph G_ρ^δ . (Color figure online)

We are now ready to prove the following result, generalisation of [16, Lemma 2], that will allow us to compute the greatest fixpoint of the operator CPre_ρ^δ :

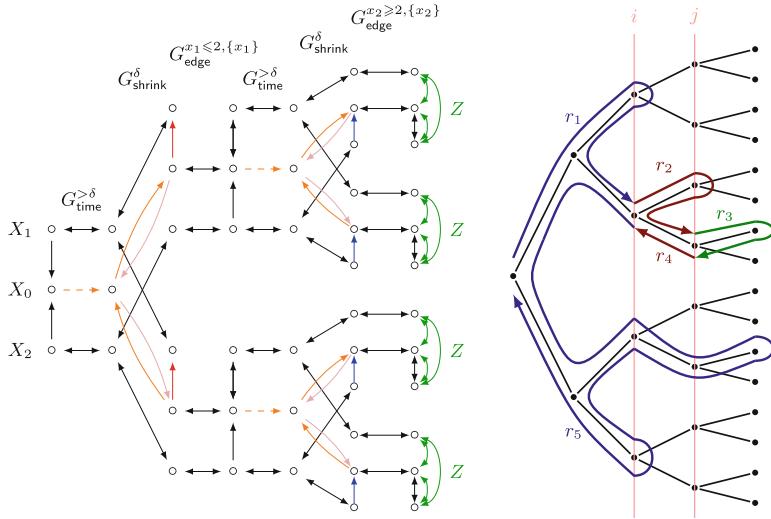


Fig. 3. On the left, the constraint graph of the path $\ell_1 \xrightarrow{x_1 \leq 2, x_1 := 0} \ell_2 \xrightarrow{x_2 \geq 2, x_2 := 0} \ell_1$. On the right, its normalised version: dashed edges have weight $(<,.)$, plain edges have weight $(\leq,.)$, black edges have weight $(.,0)$, red edges have weight $(.,2)$ and blue edges have weight $(.,-2)$.

Proposition 3. Let ρ be a path and δ be a non-negative rational number. We let $N = |\mathcal{X}_0|^2$. If $\text{CPre}_{\rho^{2N+1}}^\delta(\top) \subsetneq \text{CPre}_{\rho^{2N}}^\delta(\top)$, then $\nu X \text{CPre}_\rho^\delta(X) = \emptyset$.

Proof. Assume $\text{CPre}_{\rho^{2N+1}}^\delta(\top) \subsetneq \text{CPre}_{\rho^{2N}}^\delta(\top)$ and consider the zones $\text{CPre}_{\rho^{N+1}}^\delta(\top)$ (represented by the DBM M_1) and $\text{CPre}_{\rho^N}^\delta(\top)$ (represented by the DBM M_2). We have $M_1 \subsetneq M_2$, as otherwise the fixpoint would have already been reached after N steps. By Proposition 2, the zone corresponding to M_1 is associated with shortest paths between vertices on the left in the graph $G_{\rho^{N+1}}^\delta$. In the sequel, given a path r in this graph, $w(r)$ denotes its weight. We distinguish two cases:

Case 1: $M_1 \subsetneq M_2$ because of the rational coefficients. Then, there exists an entry $(x, y) \in \mathcal{X}_0^2$ such that $M_1[x, y] < M_2[x, y]$. The value $M_1[x, y]$ is thus associated with a shortest path between vertices X and Y in $G_{\rho^{N+1}}^\delta$. We fix a shortest path of minimal length, and denote it by r . As the entry is strictly smaller than in M_2 , this shortest path should reach the last copy of the graph G_ρ^δ . This path can be interpreted as a traversal of the binary tree of depth $|\mathcal{X}_0|^2 + 1$, reaching at least one leaf. We can prove that this entails that there exists a pair of clocks $(u, v) \in \mathcal{X}_0^2$ appearing at two levels $i < j$ of this tree, and a decomposition $r = r_1 r_2 r_3 r_4 r_5$ of the path, such that $w(r_2) + w(r_4) = (<, d)$ with $d < 0$ (Property (\dagger)). In addition, in this decomposition, r_3 is included in subgraphs of levels $k \geq j$, and the pair of paths (r_2, r_4) is called a *return path*, following the terminology of [16]. This decomposition is depicted in Fig. 2 (on the right). Intuitively, the property (\dagger) follows from the fact that as r_3 is

included in subgraphs of levels $k \geq j$, and because the final zone (on the right) is the zone \top which adds no edges, the concatenation $r' = r_1 r_3 r_5$ is also a valid path from X to Y in $G_{\rho^{N+1}}^\delta$, and is shorter than r . We conclude using the fact that r has been chosen as a shortest path of minimal weight.

Property (\dagger) allows us to prove that the greatest fixpoint is empty. Indeed, by considering iterations of ρ , one can repeat the return path associated with (r_2, r_4) and obtain paths from X to Y whose weights diverge towards $-\infty$.

Case 2: $M_1 \subsetneq M_2$ because of the ordering coefficients. We claim that this case cannot occur. Indeed, one can show that the constants will not evolve anymore after the N th iteration of the fixpoint: the coefficients can only decrease by changing from a non-strict inequality (\leq, c) to a strict one $(<, c)$. This propagation of strict inequalities is performed in at most $|\mathcal{X}_0|^2$ additional steps, thus we have $\text{CPre}_{\rho^{2N+1}}^\delta(\top) = \text{CPre}_{\rho^{2N}}^\delta(\top)$, yielding a contradiction. \square

Compared to the result of [16], the number of iterations needed before convergence grows from $|\mathcal{X}_0|^2$ to $2|\mathcal{X}_0|^2$: this is due to the presence of strict and non-strict inequalities, not considered in [16]. With the help of branching constraint graphs, we have thus shown that the greatest fixpoint can be computed in finite time: this can then be done directly with computations on zones (and not on branching constraint graphs).

Proposition 4. *Given a path ρ and a rational number δ , the greatest fixpoint $\nu X \text{CPre}_\rho^\delta(X)$ can be computed in time polynomial in $|\mathcal{X}|$ and $|\rho|$. As a consequence, one can decide whether the controller has a strategy along a lasso $\rho_1\rho_2$ in $\mathcal{G}_\delta(\mathcal{A})$ in time polynomial in $|\mathcal{X}|$ and $|\rho_1\rho_2|$.*

Solving the Robust Controller Synthesis Problem for a Lasso. We have shown how to decide whether the controller has a winning strategy for a fixed rational value of δ . We now aim at deciding whether there exists a positive value of δ for which the controller wins the game $\mathcal{G}_\delta(\mathcal{A})$ (where \mathcal{A} is restricted to a lasso $\rho_1\rho_2$). To this end, we will use a parametrised extension of DBMs, namely *shrunk DBMs*, that were introduced in [24] in order to study the parametrised state space of timed automata. Intuitively, our goal is to express *shrinkings* of guards, e.g. sets of states satisfying constraints of the form $g = 1 + \delta < x < 2 - \delta \wedge 2\delta < y$, where δ is a parameter to be chosen. Formally, a shrunk DBM is a pair (M, P) , where M is a DBM, and P is a nonnegative integer matrix called a *shrinking matrix*. This pair represents the set of valuations defined by the DBM $M - \delta P$, for any given $\delta > 0$. Considering the example g , M is the guard g obtained by setting $\delta = 0$, and P is made of the integer multipliers of δ . We adopt the following notation: when we write a statement involving a shrunk DBM (M, P) , we mean that for some $\delta_0 > 0$, the statement holds for $M - \delta P$ for all $\delta \in (0, \delta_0]$. For instance, $(M, P) = \text{Pretime}_{>\delta}(N, Q)$ means that $M - \delta P = \text{Pretime}_{>\delta}(N - \delta Q)$ for all small enough $\delta > 0$. Shrunk DBMs are closed under standard operations on zones, and as a consequence, the CPre operator can be computed on shrunk DBMs:

Lemma 2. ([25]) Let $e = (\ell, g, R, \ell')$ be an edge and (M, P) be a shrunk DBM. Then, there exists a shrunk DBM (N, Q) , that we can compute in polynomial time, such that $(N, Q) = \text{CPre}_e^\delta((M, P))$.

Proposition 5. Given a path ρ , one can compute a shrunk DBM (M, P) equal to the greatest fixpoint of the operator CPre_ρ^δ . As a consequence, one can solve the parametrised robust controller synthesis problem for a given lasso in time complexity polynomial in the number of clocks and in the length of the lasso.

Proof. The bound $2|\mathcal{X}_0|^2$ identified previously does not depend on the value of δ . Hence the algorithm for computing a shrunk DBM representing the greatest fixpoint proceeds as follows. It computes symbolically, using shrunk DBMs, the $2|\mathcal{X}_0|^2$ -th and $2|\mathcal{X}_0|^2 + 1$ -th iterations of the operator CPre_ρ^δ , from the zone \top . By monotonicity, the $2|\mathcal{X}_0|^2 + 1$ -th iteration is included in the $2|\mathcal{X}_0|^2$ -th. If the two shrunk DBMs are equal, then they are also equal to the greatest fixpoint. Otherwise, the greatest fixpoint is empty. To decide the robust controller synthesis problem for a given lasso, one first computes a shrunk DBM representing the greatest fixpoint associated with ρ_2 and, if not empty, one computes a new shrunk DBM by applying to it the operator $\text{CPre}_{\rho_1}^\delta$. Then, one checks whether the valuation $\mathbf{0}$ belongs to the resulting shrunk DBM. \square

Computing the Largest Admissible Perturbation. We say that a perturbation δ is *admissible* if the controller wins the game $\mathcal{G}_\delta(\mathcal{A})$. The parametrised robust controller synthesis problem, solved before just for a lasso, aims at deciding whether *there exists* a positive admissible perturbation. A more ambitious problem consists in determining the *largest admissible* perturbation.

The previous algorithm performs a bounded ($2|\mathcal{X}_0|^2$) number of computations of the CPre_ρ^δ operator. Instead of focusing on arbitrarily small values using shrunk DBMs as we did previously, we must perform a computation for all values of δ . To do so, we consider an extension of the (shrunk) DBMs in which each entry of the matrix (which thus represents a clock constraint) is a piecewise affine function of δ . One can observe that all the operations involved in the computation of the CPre_ρ^δ operator can be performed symbolically w.r.t. δ using piecewise affine functions. As a consequence, we obtain the following new result:

Proposition 6. We can compute the largest admissible perturbation of a lasso.

Proof. Let $\rho_1\rho_2$ be a lasso. One first computes a symbolic representation, valid for all values of δ , of the greatest fixpoint of $\text{CPre}_{\rho_2}^\delta$. To do so, one computes the $2|\mathcal{X}_0|^2$ -th and $2|\mathcal{X}_0|^2 + 1$ -th iterations of this operator, from the zone \top . We denote them by M_1 and M_2 respectively. By monotonicity, the inclusion $M_1(\delta) \subseteq M_2(\delta)$ holds for every $\delta \geq 0$. In addition, both M_1 and M_2 are decreasing w.r.t. δ , thus one can identify the value $\delta_0 = \inf\{\delta \geq 0 \mid M_1(\delta) \subsetneq M_2(\delta)\}$. Then, the greatest fixpoint is equal to M_1 for $\delta < \delta_0$, and to the emptyset for δ at least δ_0 . As a second step, one applies the operator CPre_{ρ_1} to the greatest fixpoint. We denote the result by M . To conclude, one can then compute and return the value $\sup\{\delta \in [0, \delta_0[\mid \mathbf{0} \in M(\delta)\}$ of maximal perturbation. \square

5 Synthesis of Robust Controllers

We are now ready to solve the parametrised robust controller synthesis problem, that is to find, if it exists, a lasso $\rho_1\rho_2$ and a perturbation δ such that the controller wins the game $\mathcal{G}_\delta(\mathcal{A})$ when following the lasso $\rho_1\rho_2$ as a strategy. As for the symbolic checking of emptiness of a Büchi timed language [17], we will use a double forward analysis to exhaust all possible lassos, each being tested for robustness by the techniques studied in previous section: a first forward analysis will search for ρ_1 , a path from the initial location to an accepting location, and a second forward analysis from each accepting location ℓ to find the cycle ρ_2 around ℓ . Forward analysis means that we compute the successor zone $\text{Post}_\rho(Z)$ when following path Z .

Abstractions of Lassos. Before studying in more details the two independent forward analyses, we first study what information we must keep about ρ_1 and ρ_2 in order to still being able to test the robustness of the lasso $\rho_1\rho_2$. A classical problem for robustness is the firing of a *punctual transition*, i.e. a transition where controller has a single choice of time delay: clearly such a firing will be robust for no possible choice of parameter δ . Therefore, we must at least forbid such punctual transitions in our forward analyses. We thus introduce a non-punctual successor operator $\text{Post}_\rho^{\text{NP}}$. It consists of the standard successor operator Post_ρ in the timed automaton \mathcal{A}^{NP} obtained from \mathcal{A} by making strict every constraint appearing in the guards ($1 \leq x \leq 2$ becomes $1 < x < 2$). The crucial point is that if a positive delay d can be taken by the controller while satisfying a set of strict constraints, then other delays are also possible, close enough to d . By analogy, a region is said to be *non-punctual* if it contains two valuations separated by a positive time delay. In particular, if such a region satisfies a constraint in \mathcal{A} it also satisfies the corresponding strict constraint in \mathcal{A}^{NP} . Therefore, controller wins $\mathcal{G}_\delta(\mathcal{A})$ for some $\delta > 0$ if and only if he wins $\mathcal{G}_\delta(\mathcal{A}^{\text{NP}})$ for some $\delta > 0$.

The link between non-punctuality and robustness is as follows:

Theorem 2. *Let $\rho_1\rho_2$ be a lasso of the timed automaton. We have*

$$\exists \delta > 0 \quad \mathbf{0} \in \text{CPre}_{\rho_1}^\delta(\nu X \text{CPre}_{\rho_2}^\delta(X)) \iff \text{Post}_{\rho_1}^{\text{NP}}(\mathbf{0}) \cap (\bigcup_{\delta > 0} \nu X \text{CPre}_{\rho_2}^\delta(X)) \neq \emptyset$$

Proof. The proof of this theorem relies on three main ingredients:

1. the timed automaton \mathcal{A}^{NP} allows one to compute $\bigcup_{\delta > 0} \text{CPre}_e^\delta(Z')$ by classical predecessor operator: $\text{Pre}_e^{\text{NP}}(Z') = \bigcup_{\delta > 0} \text{CPre}_e^\delta(Z')$;
2. for all edges e , and zones Z and Z' , $Z \cap \text{Pre}_e^{\text{NP}}(Z') \neq \emptyset$ if and only if $\text{Post}_e^{\text{NP}}(Z) \cap Z' \neq \emptyset$: this duality property on predecessor and successor relations always holds, in particular in \mathcal{A}^{NP} . These two ingredients already imply that the theorem holds for a path reduced to a single edge e ;
3. we then prove the theorem by induction on length of the path using that $\bigcup_{\delta > 0} \text{CPre}_{\rho_1\rho_2}^\delta(Z) = \bigcup_{\delta > 0} \text{CPre}_{\rho_1}^\delta(\bigcup_{\delta' > 0} \text{CPre}_{\rho_2}^{\delta'}(Z))$, due to the monotonicity of the $\text{CPre}_{\rho_1}^\delta$ operator. \square

Therefore, in order to test the robustness of the lasso $\rho_1\rho_2$, it is enough to only keep in memory the sets $\text{Post}_{\rho_1}^{\text{np}}(\mathbf{0})$ and $\bigcup_{\delta>0} \nu X \text{CPre}_{\rho_2}^{\delta}(X)$.

Non-punctual Forward Analysis. As a consequence of the previous theorem, we can use a classical forward analysis of the timed automaton \mathcal{A}^{np} to look for the prefix ρ_1 of the lasso $\rho_1\rho_2$. A classical inclusion check on zones allows to stop the exploration, this criterion being complete thanks to Theorem 2. It is worth reminding that we consider only bounded clocks, hence the number of reachable zones is finite, ensuring termination.

Robust Cycle Search. We now perform a second forward analysis, from each possible final location, to find a robust cycle around it. To this end, for each cycle ρ_2 , we must compute the zone $\bigcup_{\delta>0} \nu X \text{CPre}_{\rho_2}^{\delta}(X)$. This computation is obtained by arguments developed in Sect. 4 (Proposition 4). To enumerate cycles ρ_2 , we can again use a classical forward exploration, starting from the universal zone \top . Using zone inclusion to stop the exploration is not complete: considering a path ρ'_2 reaching a zone Z'_2 included in the zone Z_2 reachable using some ρ_2 , ρ'_2 could be robustly iterable while ρ_2 is not. In order to ensure termination of our analysis, we instead use reachability relations inclusion checks. These tests are performed using the technique developed in Sect. 3, based on constraint graphs (Theorem 1). The correction of this inclusion check is stated in the following lemma, where $\text{Reach}_{\rho}^{\text{np}}$ denotes the reachability relation associated with ρ in the automaton \mathcal{A}^{np} . This result is derived from the analysis based on regions in [25]. Indeed, we can prove that the non-punctual reachability relation we consider captures the existence of non-punctual aperiodic paths in the region automaton, as considered in [25].

Lemma 3. *Let ρ_1 a path from ℓ_0 to some target location ℓ_t . Let ρ_2, ρ'_2 be two paths from ℓ_t to some location ℓ , such that $\text{Reach}_{\rho_2}^{\text{np}} \subseteq \text{Reach}_{\rho'_2}^{\text{np}}$. For all paths ρ_3 from ℓ to ℓ_t , $\text{Post}_{\rho_1}^{\text{np}}(\mathbf{0}) \cap (\bigcup_{\delta>0} \nu X \text{CPre}_{\rho_2\rho_3}^{\delta}(X)) \neq \emptyset$ implies $\text{Post}_{\rho_1}^{\text{np}}(\mathbf{0}) \cap (\bigcup_{\delta>0} \nu X \text{CPre}_{\rho'_2\rho_3}^{\delta}(X)) \neq \emptyset$.*

6 Case Study

We implemented our algorithm in C++. To illustrate our approach, we present a case study on the regulation of train networks. Urban train networks in big cities are often particularly busy during rush hours: trains run in high frequency so even small delays due to incidents or passenger misbehavior can perturb the traffic and end up causing large delays. Train companies thus apply regulation techniques: they slow down or accelerate trains, and modify waiting times in order to make sure that the traffic is fluid along the network. Computing robust schedules with provable guarantees is a difficult problem (see e.g. [9]).

We study here a simplified model of a train network and aim at automatically synthesizing a controller that regulates the network despite perturbations, in order to ensure performance measures on total travel time for each train. Consider a circular train network with m stations s_0, \dots, s_{m-1} and n trains. We

require that all trains are at distinct stations at all times. There is an interval of delays $[\ell_i, u_i]$ attached to each station which bounds the travel time from s_i to $s_{i+1 \bmod m}$. Here the lower bound comes from physical limits (maximal allowed speed, and travel distance) while the upper bound comes from operator specification (e.g. it is not desirable for a train to remain at station for more than 3 min). The objective of each train i is to cycle on the network while completing each tour within a given time interval $[t_1^i, t_2^i]$.

All timing requirements are naturally encoded with clocks. Given a model, we solve the robust controller synthesis problem in order to find a controller choosing travel times for all trains ensuring a Büchi condition (visiting s_1 infinitely often). Given the fact that trains cannot be at the same station at any given time, it suffices to state the Büchi condition only for one train, since its satisfaction of the condition necessarily implies that of all other trains.

Let us present two representative instances and then comment the performance of the algorithm on a set of instances. Consider a network with two trains and m stations, with $[\ell_i, u_i] = [200, 400]$ for each station i , and the objective of both trains is the interval $[250 \cdot m, 350 \cdot m]$, that is, an average travel time between stations that lies in $[250, 350]$. The algorithm finds an accepting lasso: intuitively, by choosing δ small enough so that $m\delta < 50$, perturbations do not accumulate too much and the controller can always choose delays for both trains and satisfy the constraints. This case corresponds to scenario A in Fig. 4. Consider now the same network but with two different objectives: $[0, 300 \cdot m]$ and $[300 \cdot m, \infty)$. Thus, one train needs to complete each cycle in at most $300 \cdot m$ time units, while the other one in at least $300 \cdot m$ time units. A classical Büchi emptiness check reveals the existence of an accepting lasso: it suffices to move each train in exactly 300 time units between each station. This controller can even recover from perturbations for a bounded number of cycles: for instance, if a train arrives late at a station, the next travel time can be chosen smaller than 300. However, such corrections will cause the distance between the two trains to decrease and if such perturbations happen regularly, the system will eventually enter a deadlock. Our algorithm detects that there is no robust controller for the Büchi objective. This corresponds to the scenario B in Fig. 4.

Figure 4 summarizes the outcome of our prototype implementation on other scenarios. The tool was run on a 3.2 Ghz Intel i7 processor running Linux, with

Scenario	m	n	#Clocks	robust?	time
A	6	2	4	yes	4s
B	6	2	4	no	2s
C	6	3	5	no	263s
D	6	3	4	yes	125s
E	6	4	2	yes	53s
F	6	4	2	yes	424s
G	6	4	8		TO
H	6	4	8		TO
I	20	2	2	yes	76s
J	20	2	2	yes	55s
K	30	2	2	yes	579s

Fig. 4. Summary of experiments with different sizes. In each scenario, we assign a different objective to a subset of trains. The answer is *yes* if a robust controller was found, *no* if none exists. TO stands for a time-out of 30 min.

a 30 min time out and 2 GB of memory. The performance is sensitive to the number of clocks: on scenarios with 8 clocks the algorithm ran out of time.

7 Conclusion

Our case study illustrates the application of robust controller synthesis in small or moderate size problems. Our prototype relies on the DBM libraries that we use with twice as many clocks to store the constraints of the normalised constraint graphs. In order to scale to larger models, we plan to study extrapolation operators and their integration in the computation of reachability relations, which seems to be a challenging task. Different strategies can also be adopted for the double forward analysis, switching between the two modes using heuristics, a parallel implementation, etc.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. *Theor. Comput. Sci.* **126**(2), 183–235 (1994)
2. Bacci, G., Bouyer, P., Fahrenberg, U., Larsen, K.G., Markey, N., Reynier, P.-A.: Optimal and robust controller synthesis. In: Havelund, K., Peleska, J., Roscoe, B., de Vink, E. (eds.) FM 2018. LNCS, vol. 10951, pp. 203–221. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-95582-7_12
3. Bengtsson, J., Yi, W.: Timed automata: semantics, algorithms and tools. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) ACPN 2003. LNCS, vol. 3098, pp. 87–124. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27755-2_3
4. Berthomieu, B., Menasche, M.: An enumerative approach for analyzing time Petri nets. In: Mason, R.E.A. (ed.) Information Processing 83 - Proceedings of the 9th IFIP World Computer Congress (WCC'83), pp. 41–46. North-Holland/IFIP, September 1983
5. 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.) CONCUR 2005. LNCS, vol. 3653, pp. 66–80. Springer, Heidelberg (2005). https://doi.org/10.1007/11539452_9
6. Cassez, F., Henzinger, T.A., Raskin, J.-F.: A comparison of control problems for timed and hybrid systems. In: Tomlin, C.J., Greenstreet, M.R. (eds.) HSCC 2002. LNCS, vol. 2289, pp. 134–148. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45873-5_13
7. Chatterjee, K., Henzinger, T.A., Prabhu, V.S.: Timed parity games: complexity and robustness. In: Cassez, F., Jard, C. (eds.) FORMATS 2008. LNCS, vol. 5215, pp. 124–140. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85778-5_10
8. Comon, H., Jurski, Y.: Timed automata and the theory of real numbers. In: Baeten, J.C.M., Mauw, S. (eds.) CONCUR 1999. LNCS, vol. 1664, pp. 242–257. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48320-9_18
9. D'Ariano, A., Pranzo, M., Hansen, I.A.: Conflict resolution and train speed coordination for solving real-time timetable perturbations. *IEEE Trans. Intell. Trans. Syst.* **8**(2), 208–222 (2007)

10. Dill, D.L.: Timing assumptions and verification of finite-state concurrent systems. In: Sifakis, J. (ed.) CAV 1989. LNCS, vol. 407, pp. 197–212. Springer, Heidelberg (1990). https://doi.org/10.1007/3-540-52148-8_17
11. Henzinger, T.A., Otop, J., Samanta, R.: Lipschitz robustness of timed I/O systems. In: Jobstmann, B., Leino, K.R.M. (eds.) VMCAI 2016. LNCS, vol. 9583, pp. 250–267. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49122-5_12
12. Henzinger, T.A., Sifakis, J.: The embedded systems design challenge. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 1–15. Springer, Heidelberg (2006). https://doi.org/10.1007/11813040_1
13. Herbreteau, F., Srivathsan, B.: Efficient on-the-fly emptiness check for timed Büchi automata. In: Bouajjani, A., Chin, W.-N. (eds.) ATVA 2010. LNCS, vol. 6252, pp. 218–232. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15643-4_17
14. Herbreteau, F., Srivathsan, B., Tran, T.-T., Walukiewicz, I.: Why liveness for timed automata is hard, and what we can do about it. In: FSTTCS 2016, LIPIcs, vol. 65, pp. 48:1–48:14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2016)
15. Herbreteau, F., Srivathsan, B., Walukiewicz, I.: Efficient emptiness check for timed Büchi automata. Formal Methods Syst. Des. **40**(2), 122–146 (2012)
16. Jaubert, R., Reynier, P.-A.: Quantitative robustness analysis of flat timed automata. In: Hofmann, M. (ed.) FoSSaCS 2011. LNCS, vol. 6604, pp. 229–244. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19805-2_16
17. Laarman, A., Olesen, M.C., Dalsgaard, A.E., Larsen, K.G., van de Pol, J.: Multicore emptiness checking of timed Büchi automata using inclusion abstraction. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 968–983. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_69
18. Larsen, K.G., Legay, A., Traonouez, L.-M., Wasowski, A.: Robust synthesis for real-time systems. Theor. Comput. Sci. **515**, 96–122 (2014)
19. Li, G.: Checking timed Büchi automata emptiness using LU-abstractions. In: Ouaknine, J., Vaandrager, F.W. (eds.) FORMATS 2009. LNCS, vol. 5813, pp. 228–242. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04368-0_18
20. Prabhakar, P., Soto, M.G.: Formal synthesis of stabilizing controllers for switched systems. In: Proceedings of the 20th International Conference on Hybrid Systems: Computation and Control, HSCC 2017, New York, NY, USA, pp. 111–120. ACM (2017)
21. Prabhakar, P., Soto, M.G.: Counterexample guided abstraction refinement for stability analysis. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9779, pp. 495–512. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41528-4_27
22. Quaas, K., Shirmohammadi, M., Worrell, J.: Revisiting reachability in timed automata. In: LICS 2017. IEEE (2017)
23. Roohi, N., Prabhakar, P., Viswanathan, M.: Robust model checking of timed automata under clock drifts. In: Proceedings of the 20th International Conference on Hybrid Systems: Computation and Control, HSCC 2017, New York, NY, USA, pp. 153–162. ACM (2017)
24. Sankur, O., Bouyer, P., Markey, N.: Shrinking timed automata. In: IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2011), LIPIcs, vol. 13, pp. 90–102. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2011)
25. Sankur, O., Bouyer, P., Markey, N., Reynier, P.-A.: Robust controller synthesis in timed automata. In: D'Argenio, P.R., Melgratti, H. (eds.) CONCUR 2013. LNCS, vol. 8052, pp. 546–560. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40184-8_38

26. Tran, T.-T.: Verification of timed automata : reachability, liveness and modelling. (Vérification d'automates temporisés : sûreté, vivacité et modélisation). Ph.D. thesis, University of Bordeaux, France (2016)
27. Tripakis, S.: Checking timed Büchi automata emptiness on simulation graphs. ACM Trans. Comput. Log. **10**(3), 15:1–15:19 (2009)
28. Tripakis, S., Yovine, S., Bouajjani, A.: Checking timed Büchi automata emptiness efficiently. Formal Methods Syst. Des. **26**(3), 267–292 (2005)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Flexible Computational Pipelines for Robust Abstraction-Based Control Synthesis

Eric S. Kim^(✉), Murat Arcak, and Sanjit A. Seshia

UC Berkeley, Berkeley, CA, USA

{eskim,arcak,sseshia}@eecs.berkeley.edu

Abstract. Successfully synthesizing controllers for complex dynamical systems and specifications often requires leveraging domain knowledge as well as making difficult computational or mathematical tradeoffs. This paper presents a flexible and extensible framework for constructing robust control synthesis algorithms and applies this to the traditional abstraction-based control synthesis pipeline. It is grounded in the theory of relational interfaces and provides a principled methodology to seamlessly combine different techniques (such as dynamic precision grids, refining abstractions while synthesizing, or decomposed control predecessors) or create custom procedures to exploit an application’s intrinsic structural properties. A Dubins vehicle is used as a motivating example to showcase memory and runtime improvements.

Keywords: Control synthesis · Finite abstraction · Relational interface

1 Introduction

A control synthesizer’s high level goal is to automatically construct control software that enables a closed loop system to satisfy a desired specification. A vast and rich literature contains results that mathematically characterize solutions to different classes of problems and specifications, such as the Hamilton-Jacobi-Isaacs PDE for differential games [3], Lyapunov theory for stabilization [8], and fixed-points for temporal logic specifications [11, 17]. While many control synthesis problems have elegant mathematical solutions, there is often a gap between a solution’s theoretical characterization and the algorithms used to compute it. What data structures are used to represent the dynamics and constraints? What operations should those data structures support? How should the control synthesis algorithm be structured? Implementing solutions to the questions above can require substantial time. This problem is especially critical for computationally challenging problems, where it is often necessary to let the user *rapidly* identify and exploit structure through analysis or experimentation.

The authors were funded in part by AFOSR FA9550-18-1-0253, DARPA Assured Autonomy project, iCyPhy, Berkeley Deep Drive, and NSF grant CNS-1739816.

© The Author(s) 2019

I. Dillig and S. Tasiran (Eds.): CAV 2019, LNCS 11561, pp. 591–608, 2019.

https://doi.org/10.1007/978-3-030-25540-4_34

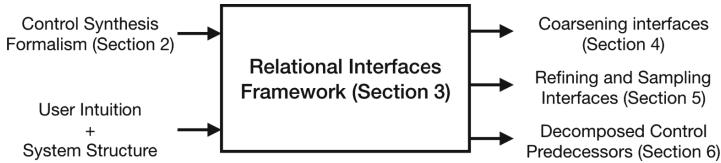


Fig. 1. By expressing many different techniques within a common framework, users are able to rapidly develop methods to exploit system structure in controller synthesis.

1.1 Bottlenecks in Abstraction-Based Control Synthesis

This paper’s goal is to enable a framework to develop extensible tools for robust controller synthesis. It was inspired in part by computational bottlenecks encountered in control synthesizers that construct finite abstractions of continuous systems, which we use as a target use case. A traditional abstraction-based control synthesis pipeline consists of three distinct stages:

1. Abstracting the continuous state system into a finite automaton whose underlying transitions faithfully mimic the original dynamics [21, 23].
2. Synthesizing a discrete controller by leveraging data structures and symbolic reasoning algorithms to mitigate combinatorial state explosion.
3. Refining the discrete controller into a continuous one. Feasibility of this step is ensured through the abstraction step.

This pipeline appears in tools PESSOA [12] and SCOTS [19], which can exhibit acute computational bottlenecks for high dimensional and nonlinear system dynamics. A common method to mitigate these bottlenecks is to exploit a specific dynamical system’s topological and algebraic properties. In MASCOT [7] and CoSyMA [14], multi-scale grids and hierarchical models capture notions of state-space locality. One could incrementally construct an abstraction of the system dynamics while performing the control synthesis step [10, 15] as implemented in tools ROCS [9] and ARCS [4]. The abstraction overhead can also be reduced by representing systems as a collection of components composed in parallel [6, 13]. These have been developed in isolation and were not previously interoperable.

1.2 Methodology

Figure 1 depicts this paper’s methodology and organization. The existing control synthesis formalism does not readily lend itself to algorithmic modifications that reflect and exploit structural properties in the system and specification. We use the theory of relational interfaces [22] as a foundation and augment it to express control synthesis pipelines. Interfaces are used to represent both system models and constraints. A small collection of atomic operators manipulates interfaces and is powerful enough to reconstruct many existing control synthesis pipelines.

One may also add new composite operators to encode desirable heuristics that exploit structural properties in the system and specifications. The last

three sections encode the techniques for abstraction-based control synthesis from Sect. 1.1 within the relational interfaces framework. By deliberately deconstructing those techniques, then reconstructing them within a compositional framework it was possible to identify implicit or unnecessary assumptions then generalize or remove them. It also makes the aforementioned techniques interoperable amongst themselves as well as future techniques.

Interfaces come equipped with a refinement partial order that formalizes when one interface abstracts another. This paper focuses on preserving the refinement relation and sufficient conditions to refine discrete controllers back to concrete ones. Additional guarantees regarding completeness, termination, precision, or decomposability can be encoded, but impose additional requirements on the control synthesis algorithm and are beyond the scope of this paper.

1.3 Contributions

To our knowledge, the application of relational interfaces to robust abstraction-based control synthesis is new. The framework’s building blocks consist of a collection of small, well understood operators that are nonetheless powerful enough to express many prior techniques. Encoding these techniques as relational interface operations forced us to simplify, formalize, or remove implicit assumptions in existing tools. The framework also exhibits numerous desirable features.

1. It enables compositional tools for control synthesis by leveraging a theoretical foundation with compositionality built into it. This paper showcases a principled methodology to seamlessly combine the methods in Sect. 1.1, as well as construct new techniques.
2. It enables a declarative approach to control synthesis by enforcing a strict separation between the high level algorithm from its low level implementation. We rely on the availability of an underlying data structure to encode and manipulate predicates. Low level predicate operations, while powerful, make it easy to inadvertently violate the refinement property. Conforming to the relational interface operations minimizes this danger.

This paper’s first half is domain agnostic and applicable to general robust control synthesis problems. The second half applies those insights to the finite abstraction approach to control synthesis. A smaller Dubins vehicle example is used to showcase and evaluate different techniques and their computational gains, compared to the unoptimized problem. In an extended version of this paper available at [1], a 6D lunar lander example leverages all techniques in this paper and introduces a few new ones.

1.4 Notation

Let $=$ be an *assertion* that two objects are mathematically equivalent; as a special case ‘ \equiv ’ is used when those two objects are sets. In contrast, the operator ‘ $==$ ’ checks whether two objects are equivalent, returning true if they are and false otherwise. A special instance of ‘ $==$ ’ is logical equivalence ‘ \Leftrightarrow ’.

Variables are denoted by lower case letters. Each variable v is associated with a domain of values $\mathcal{D}(v)$ that is analogous to the variable's type. A composite variable is a set of variables and is analogous to a bundle of wrapped wires. From a collection of variables v_1, \dots, v_M a composite variable v can be constructed by taking the union $v \equiv v_1 \cup \dots \cup v_M$ and the domain $\mathcal{D}(v) \equiv \prod_{i=1}^M \mathcal{D}(v_i)$. Note that the variables v_1, \dots, v_M above may themselves be composite. As an example if v is associated with a M -dimensional Euclidean space \mathbb{R}^M , then it is a composite variable that can be broken apart into a collection of atomic variables v_1, \dots, v_M where $\mathcal{D}(v_i) \equiv \mathbb{R}$ for all $i \in \{1, \dots, M\}$. The technical results herein do not distinguish between composite and atomic variables.

Predicates are functions that map variable assignments to a Boolean value. Predicates that stand in for expressions/formulas are denoted with capital letters. Predicates P and Q are logically equivalent (denoted by $P \Leftrightarrow Q$) if and only if $P \Rightarrow Q$ and $Q \Rightarrow P$ are true for all variable assignments. The universal and existential quantifiers \forall and \exists eliminate variables and yield new predicates. Predicates $\exists wP$ and $\forall wP$ do not depend on w . If w is a composite variable $w \equiv w_1 \cup \dots \cup w_N$ then $\exists wP$ is simply a shorthand for $\exists w_1 \dots \exists w_N P$.

2 Control Synthesis for a Motivating Example

As a simple, instructive example consider a planar Dubins vehicle that is tasked with reaching a desired location. Let $x = \{p_x, p_y, \theta\}$ be the collection of state variables, $u = \{v, \omega\}$ be a collection input variables to be controlled, $x^+ = \{p_x^+, p_y^+, \theta^+\}$ represent state variables at a subsequent time step, and $L = 1.4$ be a constant representing the vehicle length. The constraints

$$\begin{aligned} p_x^+ &== p_x + v \cos(\theta) & (F_x) \\ p_y^+ &== p_y + v \sin(\theta) & (F_y) \\ \theta^+ &== \theta + \frac{v}{L} \sin(\omega) & (F_\theta) \end{aligned}$$

characterize the discrete time dynamics. The continuous state domain is $\mathcal{D}(x) \equiv [-2, 2] \times [-2, 2] \times [-\pi, \pi]$, where the last component is periodic so $-\pi$ and π are identical values. The input domains are $\mathcal{D}(v) \equiv \{0.25, 0.5\}$ and $\mathcal{D}(\omega) \equiv \{-1.5, 0, 1.5\}$

Let predicate $F = F_x \wedge F_y \wedge F_\theta$ represent the monolithic system dynamics. Predicate T depends only on x and represents the target set $[-0.4, 0.4] \times [-0.4, 0.4] \times [-\pi, \pi]$, encoding that the vehicle's position must reach a square with any orientation. Let Z be a predicate that depends on variable x^+ that encodes a collection of states at a future time step. Equation (1) characterizes the robust controlled predecessor, which takes Z and computes the set of states from which there exists a non-blocking assignment to u that guarantees x^+ will satisfy Z , despite any non-determinism contained in F . The term $\exists x^+ F$ prevents state-control pairs from blocking, while $\forall x^+(F \Rightarrow Z)$ encodes the state-control pairs that guarantee satisfaction of Z .

$$\text{cpre}(F, Z) = \exists u (\exists x^+ F \wedge \forall x^+ (F \Rightarrow Z)). \quad (1)$$

The controlled predecessor is used to solve safety and reach games. We can solve for a region for which the target T (respectively, safe set S) can be reached (made invariant) via an iteration of an appropriate **reach** (**safe**) operator. Both iterations are given by:

$$\text{Reach Iter: } Z_0 = \perp \quad Z_{i+1} = \mathbf{reach}(F, Z_i, T) = \mathbf{cpre}(F, Z_i) \vee T. \quad (2)$$

$$\text{Safety Iter: } Z_0 = S \quad Z_{i+1} = \mathbf{safe}(F, Z_i, S) = \mathbf{cpre}(F, Z_i) \wedge S. \quad (3)$$

The above iterations are not guaranteed to reach a fixed point in a finite number of iterations, except under certain technical conditions [21]. Figure 2 depicts an approximate region where the controller can force the Dubins vehicle to enter T . We showcase different improvements relative to a base line script used to generate Fig. 2. A toolbox that adopts this paper's framework is being actively developed and is open sourced at [2]. It is written in `python 3.6` and uses the `dd` package as an interface to `CUDD` [20], a library in `C/C++` for constructing and manipulating binary decision diagrams (BDD). All experiments were run on a single core of a 2013 Macbook Pro with 2.4GHz Intel Core i7 and 8 GB of RAM.

The following section uses relational interfaces to represent the controlled predecessor $\mathbf{cpre}(\cdot)$ and iterations (2) and (3) as a computational pipeline. Subsequent sections show how modifying this pipeline leads to favorable theoretical properties and computational gains.

3 Relational Interfaces

Relational interfaces are predicates augmented with annotations about each variable's role as an input or output¹. They abstract away a component's internal implementation and only encode an input-output relation.

Definition 1 (Relational Interface [22]). An interface $M(i, o)$ consists of a predicate M over a set of input variables i and output variables o .

For an interface $M(i, o)$, we call (i, o) its input-output *signature*. An interface is a sink if it contains no outputs and has signature like (i, \emptyset) , and a source if it contains no inputs like (\emptyset, o) . Sinks and source interfaces can be interpreted as sets whereas input-output interfaces are relations. Interfaces encode relations through their predicates and can capture features such as non-deterministic outputs or

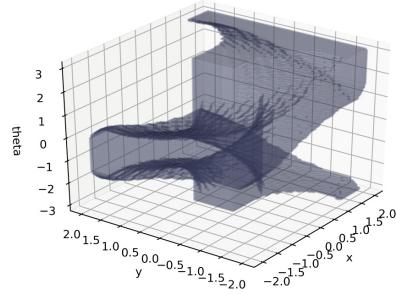


Fig. 2. Approximate solution to the Dubins vehicle reach game visualized as a subset of the state space.

¹ Relational interfaces closely resemble assume-guarantee contracts [16]; we opt to use relational interfaces because inputs and outputs play a more prominent role.

blocking (i.e., disallowed, error) inputs. A system blocks for an input assignment if there does not exist a corresponding output assignment that satisfies the interface relation. Blocking is a critical property used to declare *requirements*; sink interfaces impose constraints by modeling constrain violations as blocking inputs. Outputs on the other hand exhibit non-determinism, which is treated as an *adversary*. When one interface’s outputs are connected to another’s inputs, the outputs seek to cause blocking whenever possible.

3.1 Atomic and Composite Operators

Operators are used to manipulate interfaces by taking interfaces and variables as inputs and yielding another interface. We will show how the controlled predecessor $\text{cpre}(\cdot)$ in (1) can be constructed by composing operators appearing in [22] and one additional one. The first, output hiding, removes interface outputs.

Definition 2 (Output Hiding [22]). *Output hiding operator $\text{ohide}(w, F)$ over interface $F(i, o)$ and outputs w yields an interface with signature $(i, o \setminus w)$.*

$$\text{ohide}(w, F) = \exists w F \quad (4)$$

Existentially quantifying out w ensures that the input-output behavior over the unhidden variables is still consistent with potential assignments to w . The operator $\text{nb}(\cdot)$ is a special variant of $\text{ohide}(\cdot)$ that hides all outputs, yielding a sink encoding all non-blocking inputs to the original interface.

Definition 3 (Nonblocking Inputs Sink). *Given an interface $F(i, o)$, the nonblocking operation $\text{nb}(F)$ yields a sink interface with signature (i, \emptyset) and predicate $\text{nb}(F) = \exists o F$. If $F(i, \emptyset)$ is a sink interface, then $\text{nb}(F) = F$ yields itself. If $F(\emptyset, o)$ is a source interface, then $\text{nb}(F) = \perp$ if and only if $F \Leftrightarrow \perp$; otherwise $\text{nb}(F) = \top$.*

The interface composition operator takes multiple interfaces and “collapses” them into a single input-output interface. It can be viewed as a generalization of function composition in the special case where each interface encodes a total function (i.e., deterministic output and inputs never block).

Definition 4 (Interface Composition [22]). *Let $F_1(i_1, o_1)$ and $F_2(i_2, o_2)$ be interfaces with disjoint output variables $o_1 \cap o_2 \equiv \emptyset$ and $i_1 \cap i_2 \equiv \emptyset$ which signifies that F_2 ’s outputs may not be fed back into F_1 ’s inputs. Define new composite variables*

$$io_{12} \equiv o_1 \cap i_2 \quad (5)$$

$$i_{12} \equiv (i_1 \cup i_2) \setminus io_{12} \quad (6)$$

$$o_{12} \equiv o_1 \cup o_2. \quad (7)$$

Composition $\text{comp}(F_1, F_2)$ is an interface with signature (i_{12}, o_{12}) and predicate

$$F_1 \wedge F_2 \wedge \forall o_{12} (F_1 \Rightarrow \text{nb}(F_2)). \quad (8)$$

Interface subscripts may be swapped if instead F_2 ’s outputs are fed into F_1 .

Interfaces F_1 and F_2 are composed in parallel if $io_{21} \equiv \emptyset$ holds in addition to $io_{12} \equiv \emptyset$. Equation (8) under parallel composition reduces to $F_1 \wedge F_2$ (Lemma 6.4 in [22]) and $\text{comp}(\cdot)$ is commutative and associative. If $io_{12} \neq \emptyset$, then they are composed in series and the composition operator is only associative. Any acyclic interconnection can be composed into a single interface by systematically applying Definition 4's binary composition operator. Non-deterministic outputs are interpreted to be *adversarial*. Series composition of interfaces has a built-in notion of robustness to account for F_1 's non-deterministic outputs and blocking inputs to F_2 over the shared variables io_{12} . The term $\forall o_{12}(F_1 \Rightarrow \text{nb}(F_2))$ in Eq. (8) is a predicate over the composition's input set i_{12} . It ensures that if a potential output of F_1 may cause F_2 to block, then $\text{comp}(F_1, F_2)$ must preemptively block.

The final atomic operator is input hiding, which may only be applied to sinks. If the sink is viewed as a constraint, an input variable is “hidden” by an angelic environment that chooses an input assignment to satisfy the constraint. This operator is analogous to projecting a set into a lower dimensional space.

Definition 5 (Hiding Sink Inputs). *Input hiding operator $\text{ihide}(w, F)$ over sink interface $F(i, \emptyset)$ and inputs w yields an interface with signature $(i \setminus w, \emptyset)$.*

$$\text{ihide}(w, F) = \exists w F \quad (9)$$

Unlike the composition and output hiding operators, this operator is not included in the standard theory of relational interfaces [22] and was added to encode a controller predecessor introduced subsequently in Eq. (10).

3.2 Constructing Control Synthesis Pipelines

The robust controlled predecessor (1) can be expressed through operator composition.

Proposition 1. *The controlled predecessor operator (10) yields a sink interface with signature (x, \emptyset) and predicate equivalent to the predicate in (1).*

$$\text{cpre}(F, Z) = \text{ihide}(u, \text{ohide}(x^+, \text{comp}(F, Z))). \quad (10)$$

The simple proof is provided in the extended version at [1]. Proposition 1 signifies that controlled predecessors can be interpreted as an instance of robust composition of interfaces, followed by variable hiding. It can be shown that $\text{safe}(F, Z, S) = \text{comp}(\text{cpre}(F, Z), S)$ because $S(x, \emptyset)$ and $\text{cpre}(F, Z)$ would be composed in parallel.² Figure 3 shows a visualization of the safety game's fixed point iteration from the point of view of relational interfaces. Starting from the right-most sink interface S (equivalent to Z_0) the iteration (3) constructs a sequence of sink interfaces Z_1, Z_2, \dots encoding relevant subsets of the state space. The numerous $S(x, \emptyset)$ interfaces impose constraints and can be interpreted as monitors that raise errors if the safety constraint is violated.

² Disjunctions over sinks are required to encode $\text{reach}(\cdot)$. This will be enabled by the shared refinement operator defined in Definition 10.

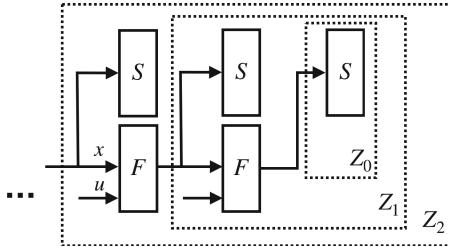


Fig. 3. Safety control synthesis iteration (3) depicted as a sequence of sink interfaces.

3.3 Modifying the Control Synthesis Pipeline

Equation (10)'s definition of $\text{cpre}(\cdot)$ is oblivious to the domains of variables x, u , and x^+ . This generality is useful for describing a problem and serving as a blank template. Whenever problem structure exists, pipeline modifications refine the general algorithm into a form that reflects the specific problem instance. They also allow a user to inject implicit preferences into a problem and reduce computational bottlenecks or to refine a solution. The subsequent sections apply this philosophy to the abstraction-based control techniques from Sect. 1.1:

- Sect. 4: Coarsening interfaces reduces the computational complexity of a problem by throwing away fine grain information. The synthesis result is conservative but the degree of conservatism can be modified.
- Sect. 5: Refining interfaces decreases result conservatism. Refinement in combination with coarsening allows one to dynamically modulate the complexity of the problem as a function of multiple criteria such as the result granularity or minimizing computational resources.
- Sect. 6: If the dynamics or specifications are decomposable then the control predecessor operator can be broken apart to reflect that decomposition.

These sections do more than simply reconstruct existing techniques in the language of relational interfaces. They uncover some implicit assumptions in existing tools and either remove them or make them explicit. Minimizing the number of assumptions ensures applicability to a diverse collection of systems and specifications and compatibility with future algorithmic modifications.

4 Interface Abstraction via Quantization

A key motivator behind abstraction-based control synthesis is that computing the game iterations from Eqs. (2) and (3) exactly is often intractable for high-dimensional nonlinear dynamics. Termination is also not guaranteed. Quantizing (or “abstracting”) continuous interfaces into a finite counterpart ensures that each predicate operation of the game terminates in finite time but at the cost of the solution’s precision. Finer quantization incurs a smaller loss of precision but

can cause the memory and computational requirements to store and manipulate the symbolic representation to exceed machine resources.

This section first introduces the notion of interface abstraction as a refinement relation. We define the notion of a quantizer and show how it is a simple generalization of many existing quantizers in the abstraction-based control literature. Finally, we show how one can inject these quantizers anywhere in the control synthesis pipeline to reduce computational bottlenecks.

4.1 Theory of Abstract Interfaces

While a controller synthesis algorithm can analyze a simpler model of the dynamics, the results have no meaning unless they can be extrapolated back to the original system dynamics. The following interface refinement condition formalizes a condition when this extrapolation can occur.

Definition 6 (Interface Refinement [22]). Let $F(i, o)$ and $\hat{F}(\hat{i}, \hat{o})$ be interfaces. \hat{F} is an abstraction of F if and only if $i \equiv \hat{i}$, $o \equiv \hat{o}$, and

$$\mathbf{nb}(\hat{F}) \Rightarrow \mathbf{nb}(F) \quad (11)$$

$$(\mathbf{nb}(\hat{F}) \wedge F) \Rightarrow \hat{F} \quad (12)$$

are valid formulas. This relationship is denoted by $\hat{F} \preceq F$.

Definition 6 imposes two main requirements between a concrete and abstract interface. Equation (11) encodes the condition where if \hat{F} accepts an input, then F must also accept it; that is, the abstract component is more aggressive with rejecting invalid inputs. Second, if both systems accept the input then the abstract output set is a superset of the concrete function's output set. The abstract interface is a conservative representation of the concrete interface because the abstraction accepts fewer inputs and exhibits more non-deterministic outputs. If both the interfaces are sink interfaces, then $\hat{F} \preceq F$ reduces down to $\hat{F} \subseteq F$ when F, \hat{F} are interpreted as sets. If both are source interfaces then the set containment direction is flipped and $\hat{F} \preceq F$ reduces down to $F \subseteq \hat{F}$.

The refinement relation satisfies the required reflexivity, transitivity, and antisymmetry properties to be a partial order [22] and is depicted in Fig. 4. This order has a bottom element \perp which is a universal abstraction. Conveniently, the bottom element \perp signifies both boolean false and the bottom of the partial order. This interface blocks for every potential input. In contrast, Boolean \top plays no special role in the partial order. While \top exhibits totally non-deterministic outputs, it also accepts inputs. A blocking input is considered “worse” than non-deterministic outputs in the refinement order. The refinement relation \preceq encodes a direction of conservatism such that any reasoning done over the abstract models is sound and can be generalized to the concrete model.

Theorem 1 (Informal Substitutability Result [22]). For any input that is allowed for the abstract model, the output behaviors exhibited by an abstract model contains the output behaviors exhibited by the concrete model.

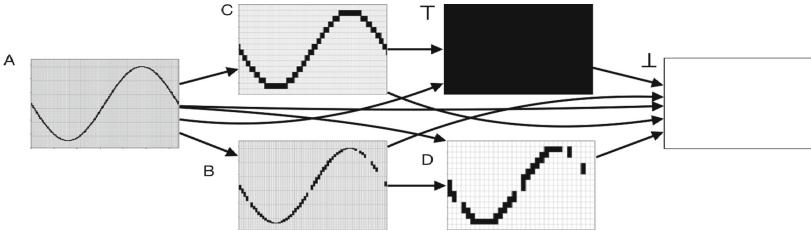


Fig. 4. Example depiction of the refinement partial order. Each small plot on the depicts input-output pairs that satisfy an interface’s predicate. Inputs (outputs) vary along the horizontal (vertical) axis. Because $B \preceq A$. Interface C exhibits more output non-determinism than A so $C \preceq A$. Similarly $D \preceq B$, $D \preceq C$, $T \preceq C$, etc. Note that B and C are incomparable because C exhibits more output non-determinism and B blocks for more inputs. The false interface \perp is a universal abstraction, while \top is incomparable with B and D .

If a property on outputs has been established for an abstract interface, then it still holds if the abstract interface is replaced with the concrete one. Informally, the abstract interface is more conservative so if a property holds with the abstraction then it must also hold for the true system. All aforementioned interface operators preserve the properties of the refinement relation of Definition 6, in the sense that they are monotone with respect to the refinement partial order.

Theorem 2 (Composition Preserves Refinement [22]). *Let $\hat{A} \preceq A$ and $\hat{B} \preceq B$. If the composition is well defined, then $\text{comp}(\hat{A}, \hat{B}) \preceq \text{comp}(A, B)$.*

Theorem 3 (Output Hiding Preserves Refinement [22]). *If $A \preceq B$, then $\text{ohide}(w, A) \preceq \text{ohide}(w, B)$ for any variable w .*

Theorem 4 (Input Hiding Preserves Refinement). *If A, B are both sink interfaces and $A \preceq B$, then $\text{ihide}(w, A) \preceq \text{ihide}(w, B)$ for any variable w .*

Proofs for Theorems 2 and 3 are provided in [22]. Theorem 4’s proof is simple and is omitted. One can think of using interface composition and variable hiding to horizontally (with respect to the refinement order) navigate the space of all interfaces. The synthesis pipeline encodes one navigated path and monotonicity of these operators yields guarantees about the path’s end point. Composite operators such as $\text{cpre}(\cdot)$ chain together multiple incremental steps. Furthermore since the composition of monotone operators is itself a monotone operator, any composite constructed from these parts is also monotone. In contrast, the coarsening and refinement operators introduced later in Definitions 8 and 10 respectively are used to move vertically and construct abstractions. The “direction” of new composite operators can easily be established through simple reasoning about the cumulative directions of their constituent operators.

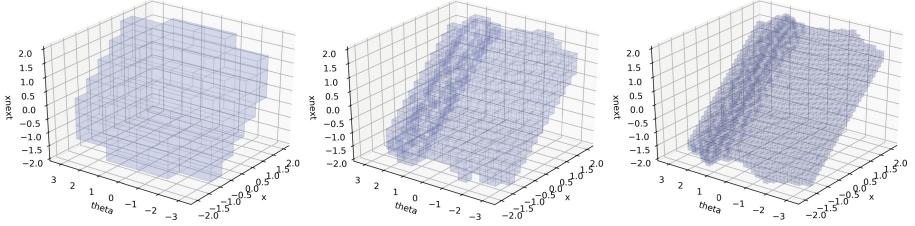


Fig. 5. Coarsening of the F_x interface to 2^3 , 2^4 and 2^5 bins along each dimension for a fixed v assignment. Interfaces are coarsened within milliseconds for BDDs but the runtime depends on the finite abstraction's data structure representation.

4.2 Dynamically Coarsening Interfaces

In practice, the sequence of interfaces Z_i generated during synthesis grows in complexity. This occurs even if the dynamics F and the target/safe sets have compact representations (i.e., fewer nodes if using BDDs). Coarsening F and Z_i combats this growth in complexity by effectively reducing the amount of information sent between iterations of the fixed point procedure.

Spatial discretization or *coarsening* is achieved by use of a quantizer interface that implicitly aggregates points in a space into a partition or cover.

Definition 7. A quantizer $Q(i, o)$ is any interface that abstracts the identity interface ($i == o$) associated with the signature (i, o) .

Quantizers decrease the complexity of the system representation and make synthesis more computationally tractable. A coarsening operator abstracts an interface by connecting it in series with a quantizer. Coarsening reduces the number of non-blocking inputs and increases the output non-determinism.

Definition 8 (Input/Output Coarsening). Given an interface $F(i, o)$ and input quantizer $Q(\hat{i}, i)$, input coarsening yields an interface with signature (\hat{i}, o) .

$$\text{icoarsen}(F, Q(\hat{i}, i)) = \text{ohide}(i, \text{comp}(Q(\hat{i}, i), F)) \quad (13)$$

Similarly, given an output quantizer $Q(o, \hat{o})$, output coarsening yields an interface with signature (i, \hat{o}) .

$$\text{oocoarsen}(F, Q(o, \hat{o})) = \text{ohide}(o, \text{comp}(F, Q(o, \hat{o}))) \quad (14)$$

Figure 5 depicts how coarsening reduces the information required to encode a finite interface. It leverages a variable precision quantizer, whose implementation is described in the extended version at [1].

The corollary below shows that quantizers can be seamlessly integrated into the synthesis pipeline while preserving the refinement order. It readily follows from Theorems 2, 3, and the quantizer definition.

Corollary 1. Input and output coarsening operations (13) and (14) are monotone operations with respect to the interface refinement order \preceq .

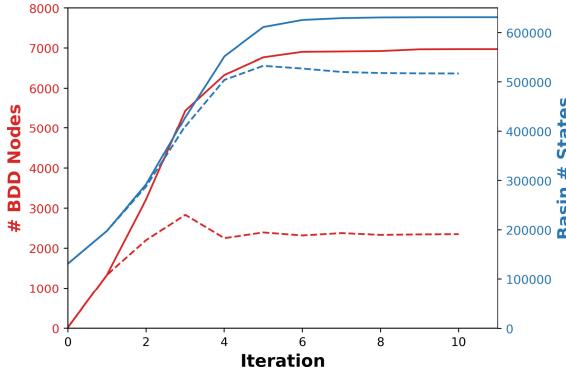


Fig. 6. Number of BDD nodes (red) and number of states in reach basin (blue) with respect to the reach game iteration with a greedy quantization. The solid lines result from the unmodified game with no coarsening heuristic. The dashed lines result from greedy coarsening whenever the winning region exceeds 3000 BDD nodes. (Color figure online)

It is difficult to know a priori where a specific problem instance lies along the spectrum between mathematical precision and computational efficiency. It is then desirable to coarsen dynamically in response to runtime conditions rather than statically beforehand. Coarsening heuristics for reach games include:

- *Downsampling with progress* [7]: Initially use coarser system dynamics to rapidly identify a coarse reach basin. Finer dynamics are used to construct a more granular set whenever the coarse iteration “stalls”. In [7] only the Z_i are coarsened during synthesis. We enable the dynamics F to be as well.
- *Greedy quantization*: Selectively coarsening along certain dimensions by checking at runtime which dimension, when coarsened, would cause Z_i to shrink the least. This reward function can be leveraged in practice because coarsening is computationally cheaper than composition. For BDDs, the winning region can be coarsened until the number of nodes reduces below a desired threshold. Figure 6 shows this heuristic being applied to reduce memory usage at the expense of answer fidelity. A fixed point is not guaranteed as long as quantizers can be dynamically inserted into the synthesis pipeline, but is once quantizers are always inserted at a fixed precision.

The most common quantizer in the literature never blocks and only increases non-determinism (such quantizers are called “strict” in [18, 19]). If a quantizer is interpreted as a partition or cover, this requirement means that the union must be equal to an entire space. Definition 7 relaxes that requirement so the union can be a subset instead. It also hints at other variants such as interfaces that don’t increase output non-determinism but instead block for more inputs.

5 Refining System Dynamics

Shared refinement [22] is an operation that takes two interfaces and merges them into a single interface. In contrast to coarsening, it makes interfaces more precise. Many tools construct system abstractions by starting from the universal abstraction \perp , then iteratively refining it with a collection of smaller interfaces that represent input-output samples. This approach is especially useful if the canonical concrete system is a black box function, Simulink model, or source code file. These representations do not readily lend themselves to the predicate operations or be coarsened directly. We will describe later how other tools implement a restrictive form of refinement that introduces unnecessary dependencies.

Interfaces can be successfully merged whenever they do not contain contradictory information. The shared refinability condition below formalizes when such a contradiction does not exist.

Definition 9 (Shared Refinability [22]). *Interfaces $F_1(i, o)$ and $F_2(i, o)$ with identical signatures are shared refinable if*

$$(\mathbf{nb}(F_1) \wedge \mathbf{nb}(F_2)) \Rightarrow \exists o(F_1 \wedge F_2) \quad (15)$$

For any inputs that do not block for all interfaces, the corresponding output sets must have a non-empty intersection. If multiple shared refinable interfaces, then they can be combined into a single one that encapsulates all of their information.

Definition 10 (Shared Refinement Operation [22]). *The shared refinement operation combines two shared refinable interfaces F_1 and F_2 , yielding a new identical signature interface corresponding to the predicate*

$$\mathbf{refine}(F_1, F_2) = (\mathbf{nb}(F_1) \vee \mathbf{nb}(F_2)) \wedge (\mathbf{nb}(F_1) \Rightarrow F_1) \wedge (\mathbf{nb}(F_2) \Rightarrow F_2). \quad (16)$$

The left term expands the set of accepted inputs. The right term signifies that if an input was accepted by multiple interfaces, the output must be consistent with each of them. The shared refinement operation reduces to disjunction for sink interfaces and to conjunction for source interfaces.

Shared refinement's effect is to move up the refinement order by combining interfaces. Given a collection of shared refinable interfaces, the shared refinement operation yields the least upper bound with respect to the refinement partial order in Definition 6. Violation of (15) can be detected if the interfaces fed into $\mathbf{refine}(\cdot)$ are not abstractions of the resulting interface.

5.1 Constructing Finite Interfaces Through Shared Refinement

A common method to construct finite abstractions is through simulation and overapproximation of forward reachable sets. This technique appears in tools such as PESSOA [12], SCOTS [19], MASCOT [7], ROCS [9] and ARCS [4]. By covering a sufficiently large portion of the interface input space, one can construct larger composite interfaces from smaller ones via shared refinement.

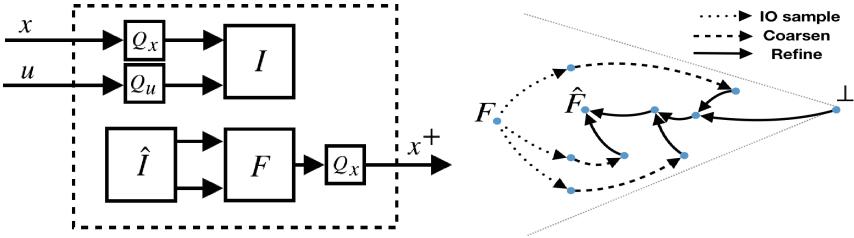


Fig. 7. (Left) Result of sample and coarsen operations for control system interface $F(x \cup u, x^+)$. The I and \hat{I} interfaces encode the same predicate, but play different roles as sink and source. (Right) Visualization of finite abstraction as traversing the refinement partial order. Nodes represent interfaces and edges signify data dependencies for interface manipulation operators. Multiple refine edges point to a single node because refinement combines multiple interfaces. Input-output (IO) sample and coarsening are unary operations so the resulting nodes only have one incoming edge. The concrete interface F refines all others, and the final result is an abstraction \hat{F} .

Smaller interfaces are constructed by sampling regions of the input space and constructing an input-output pair. In Fig. 7’s left half, a sink interface $I(x \cup u, \emptyset)$ acts as a filter. The source interface $\hat{I}(\emptyset, x \cup u)$ composed with $F(x \cup u, x^+)$ prunes any information that is outside the relevant input region. The original interface refines any sampled interface. To make samples *finite*, interface inputs and outputs are coarsened. An individual sampled abstraction is not useful for synthesis because it is restricted to a local portion of the interface input space. After sampling many finite interfaces are merged through shared refinement. The assumption $\hat{I}_i \Rightarrow \text{nb}(F)$ encodes that the dynamics won’t raise an error when simulated and is often made implicitly. Figure 7’s right half depicts the sample, coarsen, and refine operations as methods to vertically traverse the interface refinement order.

Critically, `refine()` can be called within the synthesis pipeline and does not assume that the sampled interfaces are disjoint. Figure 8 shows the results from refining the dynamics with a collection of state-control hyper-rectangles that are randomly generated via uniformly sampling their widths and offsets along each dimension. These hyper-rectangles may overlap. If the same collection of hyper-rectangles were used in MASCOT, SCOTS, ARCS, or ROCS then this would yield a much more conservative abstraction of the dynamics because their implementations are not robust to overlapping or misaligned samples. PESSOA and SCOTS circumvent this issue altogether by enforcing disjointness with an exhaustive traversal of the state-control space, at the cost of unnecessarily coupling the refinement and sampling procedures. The lunar lander in the extended version [1] embraces overlapping and uses two mis-aligned grids to construct a grid partition with p^N elements with only $p^N(\frac{1}{2})^{N-1}$ samples (where p is the number of bins along each dimension and N is the interface input dimension). This technique introduces a small degree of conservatism but its computational savings typically outweigh this cost.

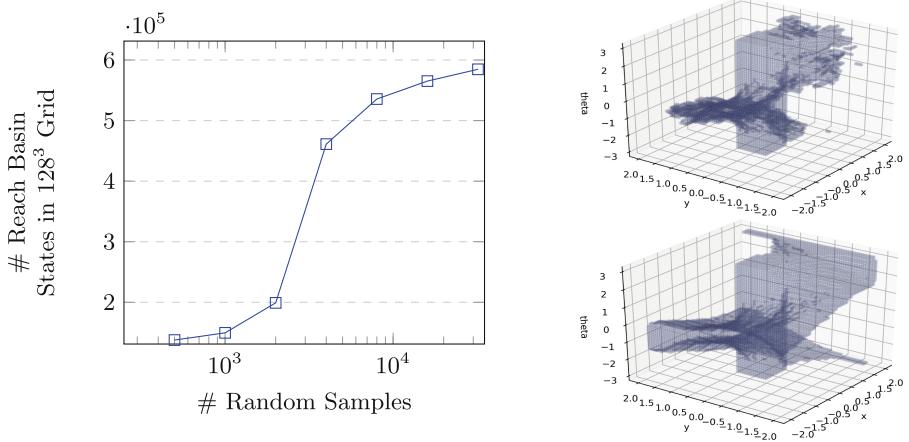


Fig. 8. The number of states in the computed reach basin grows with the number of random samples. The vertical axis is lower bounded by the number of states in the target 131k and upper bounded by 631k, the number of states using an exhaustive traversal. Naive implementations of the exhaustive traversal would require 12 million samples. The right shows basins for 3000 (top) and 6000 samples (bottom).

6 Decomposed Control Predecessor

A decomposed control predecessor is available whenever the system state space consists of a Cartesian product and the dynamics are decomposed component-wise such as F_x , F_y , and F_θ for the Dubins vehicle. This property is common for continuous control systems over Euclidean spaces. While one may construct F directly via the abstraction sampling approach, it is often intractable for larger dimensional systems. A more sophisticated approach abstracts the lower dimensional components F_x , F_y , and F_θ individually, computes $F = \text{comp}(F_x, F_y, F_\theta)$, then feeds it to the monolithic $\text{cpre}(\cdot)$ from Proposition 1. This section's approach is to avoid computing F at all and decompose the monolithic $\text{cpre}(\cdot)$. It operates by breaking apart the term $\text{ohide}(x^+, \text{comp}(F, Z))$ in such a way that it respects the decomposition structure. For the Dubins vehicle example $\text{ohide}(x^+, \text{comp}(F, Z))$ is replaced with

$$\text{ohide}(p_x^+, \text{comp}(F_x, \text{ohide}(p_y^+, \text{comp}(F_y, \text{ohide}(\theta^+, \text{comp}(F_\theta, Z))))))$$

yielding a sink interface with inputs p_x, p_y, v, θ , and ω . This representation and the original $\text{ohide}(x^+, \text{comp}(F, Z))$ are equivalent because $\text{comp}(\cdot)$ is associative and interfaces do not share outputs $x^+ \equiv \{p_x^+, p_y^+, \theta^+\}$. Figure 9 shows multiple variants of $\text{cpre}(\cdot)$ and improved runtimes when one avoids preemptively constructing the monolithic interface. The decomposed $\text{cpre}(\cdot)$ resembles techniques to exploit partitioned transition relations in symbolic model checking [5].

No tools from Sect. 1.1 natively support decomposed control predecessors. We've shown a decomposed abstraction for components composed in parallel

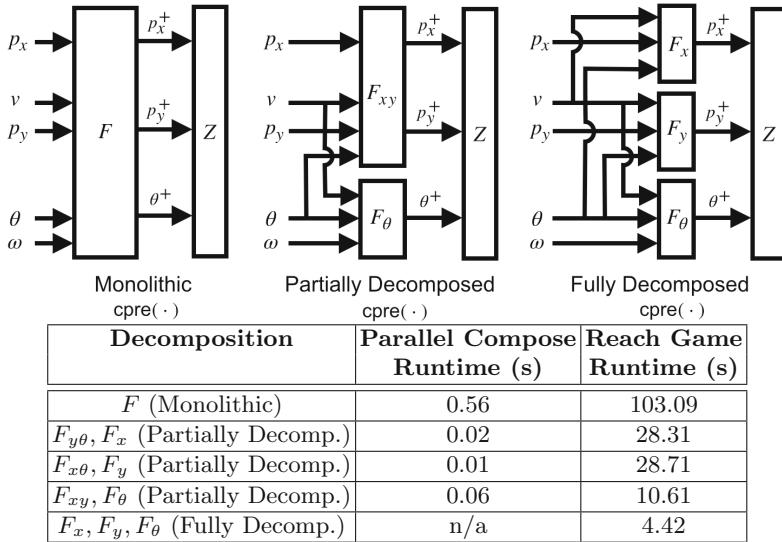


Fig. 9. A monolithic $\text{cpre}(\cdot)$ incurs unnecessary pre-processing and synthesis runtime costs for the Dubins vehicle reach game. Each variant of $\text{cpre}(\cdot)$ above composes the interfaces F_x, F_y and F_θ in different permutations. For example, F_{xy} represents $\text{comp}(F_x, F_y)$ and F represents $\text{comp}(F_x, F_y, F_\theta)$.

but this can also be generalized to series composition to capture, for example, a system where multiple components have different temporal sampling periods.

7 Conclusion

Tackling difficult control synthesis problems will require exploiting *all* available structure in a system with tools that can *flexibly adapt* to an individual problem’s idiosyncrasies. This paper lays a foundation for developing an extensible suite of interoperable techniques and demonstrates the potential computational gains in an application to controller synthesis with finite abstractions. Adhering to a simple yet powerful set of well-understood primitives also constitutes a disciplined methodology for algorithm development, which is especially necessary if one wants to develop concurrent or distributed algorithms for synthesis.

References

1. <http://arxiv.org/abs/1905.09503>
2. <https://github.com/ericskim/redax/tree/CAV19>
3. Basar, T., Olsder, G.J.: Dynamic Noncooperative Game Theory, vol. 23. Siam, Philadelphia (1999)

4. Bulancea, O.L., Nilsson, P., Ozay, N.: Nonuniform abstractions, refinement and controller synthesis with novel BDD encodings. CoRR, [arXiv: abs/1804.04280](https://arxiv.org/abs/abs/1804.04280) (2018)
5. Burch, J., Clarke, E., Long, D.: Symbolic model checking with partitioned transition relations (1991)
6. Gruber, F., Kim, E., Arcak, M.: Sparsity-aware finite abstraction. In: 2017 IEEE 56th Conference on Decision and Control (CDC), December 2017
7. Hsu, K., Majumdar, R., Mallik, K., Schmuck, A.-K.: Multi-layered abstraction-based controller synthesis for continuous-time systems. In: Proceedings of the 21st International Conference on Hybrid Systems: Computation and Control (Part of CPS Week), HSCC 2018, pp. 120–129. ACM, New York (2018)
8. Khalil, H.K., Grizzle, J.W.: Nonlinear Systems, vol. 3. Prentice Hall, Upper Saddle River, New Jersey (2002)
9. Li, Y., Liu, J.: ROCS: a robustly complete control synthesis tool for nonlinear dynamical systems. In: Proceedings of the 21st International Conference on Hybrid Systems: Computation and Control (Part of CPS Week), HSCC 2018, pp. 130–135. ACM, New York (2018)
10. Liu, J.: Robust abstractions for control synthesis: completeness via robustness for linear-time properties. In: Proceedings of the 20th International Conference on Hybrid Systems: Computation and Control, HSCC 2017, pp. 101–110. ACM, New York (2017)
11. Majumdar, R.: Symbolic algorithms for verification and control. Ph.D. thesis, University of California, Berkeley (2003)
12. Mazo Jr., M., Davitian, A., Tabuada, P.: PESSOA: a tool for embedded controller synthesis. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 566–569. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_49
13. Meyer, P.J., Girard, A., Witrant, E.: Compositional abstraction and safety synthesis using overlapping symbolic models. IEEE Trans. Autom. Control. **63**, 1835–1841 (2017)
14. Mouelhi, S., Girard, A., Gössler, G.: CoSyMA: a tool for controller synthesis using multi-scale abstractions. In: 16th International Conference on Hybrid Systems: Computation and Control, pp. 83–88. ACM (2013)
15. Nilsson, P., Ozay, N., Liu, J.: Augmented finite transition systems as abstractions for control synthesis. Discret. Event Syst. **27**(2), 301–340 (2017)
16. Nuzzo, P.: Compositional design of cyber-physical systems using contracts. Ph.D. thesis, EECS Department, University of California, Berkeley, August 2015
17. Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of reactive(1) designs. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 364–380. Springer, Heidelberg (2005). https://doi.org/10.1007/11609773_24
18. Reißig, G., Weber, A., Rungger, M.: Feedback refinement relations for the synthesis of symbolic controllers. IEEE Trans. Autom. Control. **62**(4), 1781–1796 (2017)
19. Rungger, M., Zamani, M.: SCOTS: a tool for the synthesis of symbolic controllers. In: 19th International Conference on Hybrid Systems: Computation and Control, pp. 99–104. ACM (2016)
20. Somenzi, F.: CUDD: CU Decision Diagram Package. <http://vlsi.colorado.edu/~fabio/CUDD/>, Version 3.0.0 (2015)
21. Tabuada, P.: Verification and Control of Hybrid Systems. Springer, New York (2009). <https://doi.org/10.1007/978-1-4419-0224-5>

22. Tripakis, S., Lickly, B., Henzinger, T.A., Lee, E.A.: A theory of synchronous relational interfaces. ACM Trans. Program. Lang. Syst. (TOPLAS) **33**(4), 14 (2011)
23. Zamani, M., Pola, G., Mazo, M., Tabuada, P.: Symbolic models for nonlinear control systems without stability assumptions. IEEE Trans. Autom. Control **57**(7), 1804–1809 (2012)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Temporal Stream Logic: Synthesis Beyond the Bools

Bernd Finkbeiner¹, Felix Klein¹⁽),
Ruzica Piskac²,
and Mark Santolucito²



¹ Saarland University, Saarbrücken, Germany

klein@react.uni-saarland.de

² Yale University, New Haven, USA

Abstract. Reactive systems that operate in environments with complex data, such as mobile apps or embedded controllers with many sensors, are difficult to synthesize. Synthesis tools usually fail for such systems because the state space resulting from the discretization of the data is too large. We introduce TSL, a new temporal logic that separates control and data. We provide a CEGAR-based synthesis approach for the construction of implementations that are guaranteed to satisfy a TSL specification for all possible instantiations of the data processing functions. TSL provides an attractive trade-off for synthesis. On the one hand, synthesis from TSL, unlike synthesis from standard temporal logics, is undecidable in general. On the other hand, however, synthesis from TSL is scalable, because it is independent of the complexity of the handled data. Among other benchmarks, we have successfully synthesized a music player Android app and a controller for an autonomous vehicle in the Open Race Car Simulator (TORCS).

1 Introduction

In reactive synthesis, we automatically translate a formal specification, typically given in a temporal logic, into a controller that is guaranteed to satisfy the specification. Over the past two decades there has been much progress on reactive synthesis, both in terms of algorithms, notably with techniques like GR(1)-synthesis [7] and bounded synthesis [20], and in terms of tools, as showcased, for example, in the annual SYNTCOMP competition [25].

In practice however, reactive synthesis has seen limited success. One of the largest published success stories [6] is the synthesis of the AMBA bus protocol. To push synthesis even further, automatically synthesizing a controller for

Supported by the European Research Council (ERC) Grant OSARES (No. 683300), the German Research Foundation (DFG) as part of the Collaborative Research Center Foundations of Perspicuous Software Systems (TRR 248, 389792660), and the National Science Foundation (NSF) Grant CCF-1302327.

© The Author(s) 2019

I. Dillig and S. Tasiran (Eds.): CAV 2019, LNCS 11561, pp. 609–629, 2019.

https://doi.org/10.1007/978-3-030-25540-4_35

an autonomous system has been recognized to be of critical importance [52]. Despite many years of experience with synthesis tools, our own attempts to synthesize such controllers with existing tools have been unsuccessful. The reason is that the tools are unable to handle the data complexity of the controllers. The controller only needs to switch between a small number of behaviors, like steering during a bend, or shifting gears on high rpm. The number of control states in a typical controller (cf. [18]) is thus not much different from the arbiter in the AMBA case study. However, in order to correctly initiate transitions between control states, the driving controller must continuously process data from more than 20 sensors.

If this data is included (even as a rough discretization) in the state space of the controller, then the synthesis problem is much too large to be handled by any available tools. It seems clear then, that a scalable synthesis approach must separate control and data. If we assume that the data processing is handled by some other approach (such as deductive synthesis [38] or manual programming), is it then possible to solve the remaining reactive synthesis problem?

In this paper, we show scalable reactive synthesis is indeed possible. Separating data and control has allowed us to synthesize reactive systems, including an autonomous driving controller and a music player app, that had been impossible to synthesize with previously available tools. However, the separation of data and control implies some fundamental changes to reactive synthesis, which we describe in the rest of the paper. The changes also imply that the reactive synthesis problem is no longer, in general, decidable. We thus trade theoretical decidability for practical scalability, which is, at least with regard to the goal of synthesizing realistic systems, an attractive trade-off.

We introduce Temporal Stream Logic (TSL), a new temporal logic that includes *updates*, such as $\llbracket y \leftarrow f x \rrbracket$, and predicates over arbitrary function terms. The update $\llbracket y \leftarrow f x \rrbracket$ indicates that the result of applying function f to variable x is assigned to y . The implementation of predicates and functions is not part of the synthesis problem. Instead, we look for a system that satisfies the TSL specification *for all possible interpretations of the functions and predicates*.

This implicit quantification over all possible interpretations provides a useful abstraction: it allows us to *independently* implement the data processing part. On the other hand, this quantification is also the reason for the undecidability of the synthesis problem. If a predicate is applied to the same term *twice*, it must (independently of the interpretation) return the *same* truth value. The synthesis must then implicitly maintain a (potentially infinite) set of terms to which the predicate has previously been applied. As we show later, this set of terms can be used to encode PCP [45] for a proof of undecidability.

We present a practical synthesis approach for TSL specifications, which is based on bounded synthesis [20] and counterexample-guided abstraction refinement (CEGAR) [9]. We use bounded synthesis to search for an implementation up to a (iteratively growing) bound on the number of states. This approach underapproximates the actual TSL synthesis problem by leaving the interpretation of the predicates to the environment. The underapproximation allows

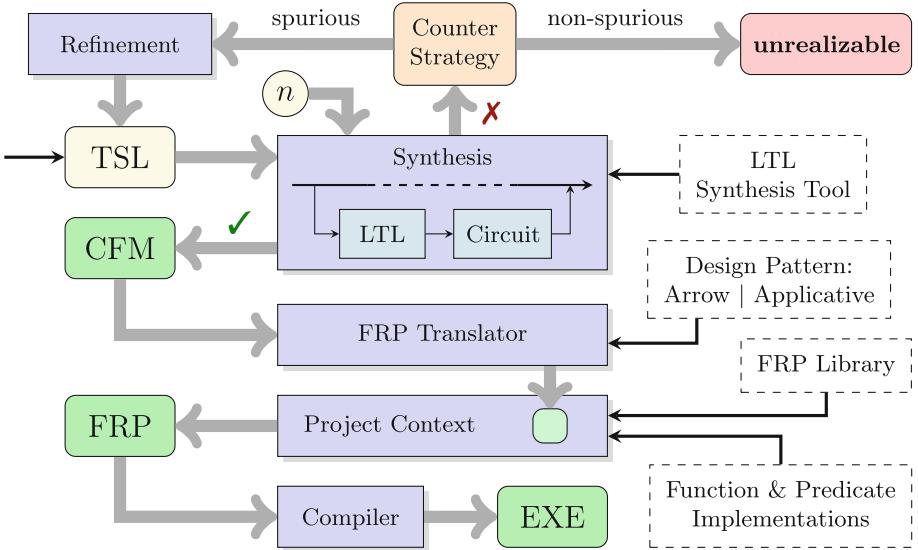


Fig. 1. The TSL synthesis procedure uses a modular design. Each step takes input from the previous step as well as interchangeable modules (dashed boxes).

for inconsistent behaviors: the environment might assign different truth values to the same predicate when evaluated at different points in time, even if the predicate is applied to the same term. However, if we find an implementation in this underapproximation, then the CEGAR loop terminates and we have a correct implementation for the original TSL specification. If we do not find an implementation in the underapproximation, we compute a counter strategy for the environment. Because bounded synthesis reduces the synthesis problem to a safety game, the counter strategy is a reachability strategy that can be represented as a finite tree. We check whether the counter strategy is spurious by searching for a pair of positions in the strategy where some predicate results in different truth values when applied to the same term. If the counter strategy is not spurious, then no implementation exists for the considered bound, and we increase the bound. If the counter strategy is spurious, then we introduce a constraint into the specification that eliminates the incorrect interpretation of the predicate, and continue with the refined specification.

A general overview of this procedure is shown in Fig. 1. The top half of the figure depicts the bounded search for an implementation that realizes a TSL specification using the CEGAR loop to refine the specification. If the specification is realizable, we proceed in the bottom half of the process, where a synthesized implementation is converted to a control flow model (CFM) determining the control of the system. We then specialize the CFM to Functional Reactive Programming (FRP), which is a popular and expressive programming paradigm for building reactive programs using functional programming languages [14].

Sys.leaveApp() :	ALWAYS $(\text{leaveApp Sys} \wedge \text{musicPlaying MP} \rightarrow [\text{Ctrl} \leftarrow \text{pause}()])$
if (MP.musicPlaying()) Ctrl.pause()	ALWAYS $(\text{resumeApp Sys} \rightarrow [\text{Ctrl} \leftarrow \text{play Tr (trackPos MP)}])$

Fig. 2. Sample code and specification for the music player app.

Our framework supports any FRP library using the *Arrow* or *Applicative* design patterns, which covers most of the existing FRP libraries (e.g. [2, 3, 10, 41]). Finally, the synthesized control flow is embedded into a project context, where it is equipped with function and predicate implementations and then compiled to an executable program.

Our experience with synthesizing systems based on TSL specifications has been extremely positive. The synthesis works for a broad range of benchmarks, ranging from classic reactive synthesis problems (like escalator control), through programming exercises from functional reactive programming, to novel case studies like our music player app and the autonomous driving controller for a vehicle in the Open Race Car Simulator (TORCS).

2 Motivating Example

To demonstrate the utility of our method, we synthesized a music player Android app¹ from a TSL specification. A major challenge in developing Android apps is the temporal behavior of an app through the *Android lifecycle* [46]. The Android lifecycle describes how an app should handle being paused, when moved to the background, coming back into focus, or being terminated. In particular, *resume and restart errors* are commonplace and difficult to detect and correct [46]. Our music player app demonstrates a situation in which a resume and restart error could be unwittingly introduced when programming by hand, but is avoided by providing a specification. We only highlight the key parts of the example here to give an intuition of TSL. The complete specification is presented in [19].

Our music player app utilizes the Android music player library (MP), as well as its control interface (**Ctrl**). It pauses any playing music when moved to the background (for instance if a call is received), and continues playing the currently selected track (Tr) at the last track position when the app is resumed. In the Android system (Sys), the `leaveApp` method is called whenever the app moves to the background, while the `resumeApp` method is called when the app is brought back to the foreground. To avoid confusion between pausing music and pausing the app, we use `leaveApp` and `resumeApp` in place of the Android methods

¹ <https://play.google.com/store/apps/details?id=com.mark.myapplication>.

<pre> bool wasPlaying = false Sys.leaveApp() : if (MP.musicPlaying()) : wasPlaying = true Ctrl.pause() else wasPlaying = false Sys.resumeApp() : if (wasPlaying) pos = MP.trackPos() Ctrl.play(Tr, pos) </pre>	$\text{ALWAYS } \left((leaveApp \text{ Sys} \wedge \text{musicPlaying MP} \rightarrow [\![\text{Ctrl} \leftarrow \text{pause}()]\!]) \wedge ([\![\text{Ctrl} \leftarrow \text{play Tr} (\text{trackPos MP})]\!] \wedge \text{AS_SOON_AS resumeApp Sys}) \right)$
--	---

Fig. 3. The effect of a minor change in functionality on code versus a specification.

onPause and onResume. A programmer might manually write code for this as shown on the left in Fig. 2.

The behavior of this can be directly described in TSL as shown on the right in Fig. 2. Even eliding a formal introduction of the notation for now, the specification closely matches the textual specification. First, when the user leaves the app and the music is playing, the music pauses. Likewise for the second part, when the user resumes the app, the music starts playing again.

However, assume we want to change the behavior so that the music only plays on resume when the music had been playing before leaving the app in the first place. In the manually written program, this new functionality requires an additional variable `wasPlaying` to keep track of the music state. Managing the state requires multiple changes in the code as shown on the left in Fig. 3. The required code changes include: a conditional in the `resumeApp` method, setting `wasPlaying` appropriately in two places in `leaveApp`, and providing an initial value. Although a small example, it demonstrates how a minor change in functionality may require wide-reaching code changes. In addition, this change introduces a globally scoped variable, which then might accidentally be set or read elsewhere. In contrast, it is a simple matter to change the TSL specification to reflect this new functionality. Here, we only update one part of the specification to say that if the user leaves the app and the music is playing, the music has to play again as soon as the app resumes.

Synthesis allows us to specify a temporal behavior without worrying about the implementation details. In this example, writing the specification in TSL has eliminated the need of an additional state variable, similarly to a higher order `map` eliminating the need for an iteration variable. However, in more complex examples the benefits compound, as TSL provides a modular interface to specify behaviors, offloading the management of multiple interconnected temporal behaviors from the user to the synthesis engine.

3 Preliminaries

We assume time to be discrete and denote it by the set \mathbb{N} of positive integers. A value is an arbitrary object of arbitrary type. \mathcal{V} denotes the set of all values. The Boolean values are denoted by $\mathcal{B} \subseteq \mathcal{V}$. A stream $s: \mathbb{N} \rightarrow \mathcal{V}$ is a function fixing values at each point in time. An n -ary function $f: \mathcal{V}^n \rightarrow \mathcal{V}$ determines new values from n given values, where the set of all functions (of arbitrary arity) is given by \mathcal{F} . Constants are functions of arity 0. Every constant is a value, i.e., is an element of $\mathcal{F} \cap \mathcal{V}$. An n -ary predicate $p: \mathcal{V}^n \rightarrow \mathcal{B}$ checks a property over n values. The set of all predicates (of arbitrary arity) is given by \mathcal{P} , where $\mathcal{P} \subseteq \mathcal{F}$. We use $B^{[A]}$ to denote the set of all total functions with domain A and image B .

In the classical synthesis setting, inputs and outputs are vectors of Booleans, where the standard abstraction treats inputs and outputs as atomic propositions $\mathcal{I} \cup \mathcal{O}$, while their Boolean combinations form an alphabet $\Sigma = 2^{\mathcal{I} \cup \mathcal{O}}$. Behavior then is described through infinite sequences $\alpha = \alpha(0)\alpha(1)\alpha(2)\dots \in \Sigma^\omega$. A *specification* describes a relation between input sequences $\alpha \in (2^{\mathcal{I}})^\omega$ and output sequences $\beta \in (2^{\mathcal{O}})^\omega$. Usually, this relation is not given by explicit sequences, but by a formula in a temporal logic. The most popular such logic is Linear Temporal Logic (LTL) [43], which uses Boolean connectives to specify behavior at specific points in time, and temporal connectives, to relate sub-specifications over time. The realizability and synthesis problems for LTL are 2ExPTIME-complete [44].

An implementation describes a realizing strategy, formalized via infinite trees. A Φ -labeled and Υ -branching tree is a function $\sigma: \Upsilon^* \rightarrow \Phi$, where Υ denotes the set of branching directions along a tree. Every node of the tree is given by a finite prefix $v \in \Upsilon^*$, which fixes the path to reach a node from the root. Every node is labeled by an element of Φ . For infinite paths $\nu \in \Upsilon^\omega$, the branch $\sigma|\nu$ denotes the sequence of labels that appear on ν , i.e., $\forall t \in \mathbb{N}. (\sigma|\nu)(t) = \sigma(\nu(0)\dots\nu(t-1))$.

4 Temporal Stream Logic

We present a new logic: Temporal Stream Logic (TSL), which is especially designed for synthesis and allows for the manipulation of infinite streams of arbitrary (even non-enumerative, or higher order) type. It provides a straightforward notation to specify how outputs are computed from inputs, while using an intuitive interface to access time. The main focus of TSL is to describe temporal control flow, while abstracting away concrete implementation details. This not only keeps the logic intuitive and simple, but also allows a user to identify problems in the control flow even without a concrete implementation at hand. In this way, the use of TSL scales up to any required abstraction, such as API calls or complex algorithmic transformations.

Architecture. A TSL formula φ specifies a reactive system that in every time step processes a finite number of inputs \mathbb{I} and produces a finite number of outputs \mathbb{O} . Furthermore, it uses cells \mathbb{C} to store a value computed at time t , which can then be reused in the next time step $t+1$. An overview of the architecture of such a system is given in Fig. 4a. In terms of behavior, the environment produces infinite

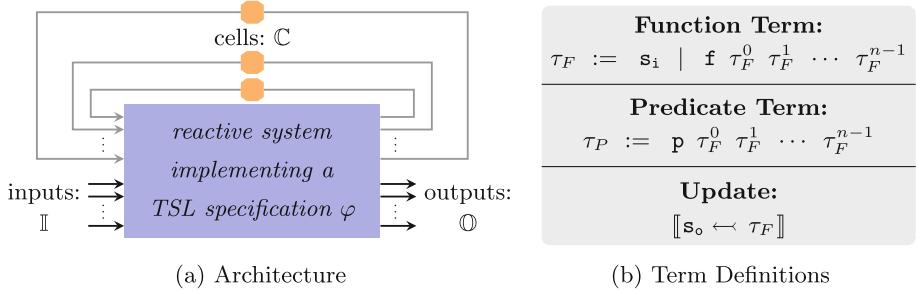


Fig. 4. General architecture of reactive systems that are specified in TSL on the left, and the structure of function, predicate and updates on the right.

streams of input data, while the system uses pure (side-effect free) functions to transform the values of these input streams in every time step. After their transformation, the data values are either passed to an output stream or are passed to a cell, which pipes the output value from one time step back to the corresponding input value of the next. The behaviour of the system is captured by its infinite execution over time.

Function Terms, Predicate Terms, and Updates. In TSL we differentiate between two elements: we use purely functional transformations, reflected by functions $f \in \mathcal{F}$ and their compositions, and predicates $p \in \mathcal{P}$, used to control how data flows inside the system. To argue about both elements we use a term based notation, where we distinguish between function terms τ_F and predicate terms τ_P , respectively. Function terms are either constructed from inputs or cells ($s_i \in \mathbb{I} \cup \mathbb{C}$), or from functions, recursively applied to a set of function terms. Predicate terms are constructed similarly, by applying a predicate to a set of function terms. Finally, an update takes the result of a function computation and passes it either to an output or a cell ($s_o \in \mathbb{O} \cup \mathbb{C}$). An overview of the syntax of the different term notations is given in Fig. 4b. Note that we use curried argument notation similar to functional programming languages.

We denote sets of function and predicate terms, and updates by \mathcal{T}_F , \mathcal{T}_P and \mathcal{T}_U , respectively, where $\mathcal{T}_P \subseteq \mathcal{T}_F$. We use \mathbb{F} to denote the set of function literals and $\mathbb{P} \subseteq \mathbb{F}$ to denote the set of predicate literals, where the literals s_i , s_o , f and p are symbolic representations of inputs and cells, outputs and cells, functions, and predicates, respectively. Literals are used to construct terms as shown in Fig. 4b. Since we use a symbolic representation, functions and predicates are not tied to a specific implementation. However, we still classify them according to their arity, i.e., the number of function terms they are applied to, as well as by their type: input, output, cell, function or predicate. Furthermore, terms can be compared syntactically using the equivalence relation \equiv . To assign a semantic interpretation to functions, we use an assignment function $\langle \cdot \rangle : \mathbb{F} \rightarrow \mathcal{F}$.

Inputs, Outputs, and Computations. We consider momentary inputs $i \in \mathcal{V}^{\mathbb{I}}$, which are assignments of inputs $\mathbf{i} \in \mathbb{I}$ to values $v \in \mathcal{V}$. For the sake of readability let $\mathcal{I} = \mathcal{V}^{\mathbb{I}}$. Input streams are infinite sequences $\iota \in \mathcal{I}^\omega$ consisting of infinitely many momentary inputs.

Similarly, a momentary output $o \in \mathcal{V}^{[\mathbb{O}]}$ is an assignment of outputs $\mathbf{o} \in \mathbb{O}$ to values $v \in \mathcal{V}$, where we also use $\mathcal{O} = \mathcal{V}^{[\mathbb{O}]}$. Output streams are infinite sequences $\varrho \in \mathcal{O}^\omega$. To capture the behavior of a cell, we introduce the notion of a computation ς . A computation fixes the function terms that are used to compute outputs and cell updates, without fixing semantics of function literals. Intuitively, a computation only determines which function terms are used to compute an output, but abstracts from actually computing it.

The basic element of a computation is a computation step $c \in \mathcal{T}_F^{[\mathbb{OUC}]}$, which is an assignment of outputs and cells $\mathbf{s}_i \in \mathbb{O} \cup \mathbb{C}$ to function terms $\tau_F \in \mathcal{T}_F$. For the sake of readability let $\mathcal{C} = \mathcal{T}_F^{[\mathbb{OUC}]}$. A computation step fixes the control flow behaviour at a single point in time. A computation $\varsigma \in \mathcal{C}^\omega$ is an infinite sequence of computation steps.

As soon as input streams, and function and predicate implementations are known, computations can be turned into output streams. To this end, let $\langle \cdot \rangle : \mathbb{F} \rightarrow \mathcal{F}$ be some function assignment. Furthermore, assume that there are predefined constants $init_c \in \mathcal{F} \cap \mathcal{V}$ for every cell $c \in \mathbb{C}$, which provide an initial value for each stream at the initial point in time. To receive an output stream from a computation $\varsigma \in \mathcal{C}^\omega$ under the input stream ι , we use an evaluation function $\eta_{\langle \cdot \rangle} : \mathcal{C}^\omega \times \mathcal{I}^\omega \times \mathbb{N} \times \mathcal{T}_F \rightarrow \mathcal{V}$:

$$\eta_{\langle \cdot \rangle}(\varsigma, \iota, t, \mathbf{s}_i) = \begin{cases} \iota(t)(\mathbf{s}_i) & \text{if } \mathbf{s}_i \in \mathbb{I} \\ init_{\mathbf{s}_i} & \text{if } \mathbf{s}_i \in \mathbb{C} \wedge t = 0 \\ \eta_{\langle \cdot \rangle}(\varsigma, \iota, t - 1, \varsigma(t - 1)(\mathbf{s}_i)) & \text{if } \mathbf{s}_i \in \mathbb{C} \wedge t > 0 \end{cases}$$

$$\eta_{\langle \cdot \rangle}(\varsigma, \iota, t, \mathbf{f} \tau_0 \dots \tau_{m-1}) = \langle \mathbf{f} \rangle \eta_{\langle \cdot \rangle}(\varsigma, \iota, t, \tau_0) \dots \eta_{\langle \cdot \rangle}(\varsigma, \iota, t, \tau_{m-1})$$

Then $\varrho_{\langle \cdot \rangle, \varsigma, \iota} \in \mathcal{O}^\omega$ is defined via $\varrho_{\langle \cdot \rangle, \varsigma, \iota}(t)(\mathbf{o}) = \eta_{\langle \cdot \rangle}(\varsigma, \iota, t, \mathbf{o})$ for all $t \in \mathbb{N}$, $\mathbf{o} \in \mathbb{O}$.

Syntax. Every TSL formula φ is built according to the following grammar:

$$\varphi := \tau \in \mathcal{T}_P \cup \mathcal{T}_U \mid \neg \varphi \mid \varphi \wedge \varphi \mid \circlearrowleft \varphi \mid \vartheta \mathcal{U} \psi$$

An atomic proposition τ consists either of a predicate term, serving as a Boolean interface to the inputs, or of an update, enforcing a respective flow at the current point in time. Next, we have the Boolean operations via negation and conjunction, that allow us to express arbitrary Boolean combinations of predicate evaluations and updates. Finally, we have the temporal operator next: $\circlearrowleft \psi$, to specify the behavior at the next point in time and the temporal operator until: $\vartheta \mathcal{U} \psi$, which enforces a property ϑ to hold until the property ψ holds, where ψ must hold at some point in the future eventually.

Semantics. Formally, this leads to the following semantics. Let $\langle \cdot \rangle : \mathbb{F} \rightarrow \mathcal{F}$, $\iota \in \mathcal{I}^\omega$, and $\varsigma \in \mathcal{C}^\omega$ be given, then the validity of a TSL formula φ with respect to ς and ι is defined inductively over $t \in \mathbb{N}$ via:

$$\begin{aligned}\varsigma, \iota, t \models_{\langle \cdot \rangle} p \tau_0 \cdots \tau_{m-1} &\Leftrightarrow \eta_{\langle \cdot \rangle}(\varsigma, \iota, t, p \tau_0 \cdots \tau_{m-1}) \\ \varsigma, \iota, t \models_{\langle \cdot \rangle} [\![s \leftarrow \tau]\!] &\Leftrightarrow \varsigma(t)(s) \equiv \tau \\ \varsigma, \iota, t \models_{\langle \cdot \rangle} \neg \psi &\Leftrightarrow \varsigma, \iota, t \not\models_{\langle \cdot \rangle} \psi \\ \varsigma, \iota, t \models_{\langle \cdot \rangle} \vartheta \wedge \psi &\Leftrightarrow \varsigma, \iota, t \models_{\langle \cdot \rangle} \vartheta \wedge \varsigma, \iota, t \models_{\langle \cdot \rangle} \psi \\ \varsigma, \iota, t \models_{\langle \cdot \rangle} \circ \psi &\Leftrightarrow \varsigma, \iota, t + 1 \models_{\langle \cdot \rangle} \psi \\ \varsigma, \iota, t \models_{\langle \cdot \rangle} \vartheta \mathcal{U} \psi &\Leftrightarrow \exists t'' \geq t. \forall t' \leq t' < t''. \varsigma, \iota, t' \models_{\langle \cdot \rangle} \vartheta \wedge \varsigma, \iota, t'' \models_{\langle \cdot \rangle} \psi\end{aligned}$$

Consider that the satisfaction of a predicate depends on the current computation step and the steps of the past, while for updates it only depends on the current computation step. Furthermore, updates are only checked syntactically, while the satisfaction of predicates depends on the given assignment $\langle \cdot \rangle$ and the input stream ι . We say that ς and ι satisfy φ , denoted by $\varsigma, \iota \models_{\langle \cdot \rangle} \varphi$, if $\varsigma, \iota, 0 \models_{\langle \cdot \rangle} \varphi$.

Beside the basic operators, we have the standard derived Boolean operators, as well as the derived temporal operators: *release* $\varphi \mathcal{R} \psi \equiv \neg((\neg \psi) \mathcal{U} (\neg \varphi))$, *finally* $\diamond \varphi \equiv \text{true} \mathcal{U} \varphi$, *always* $\square \varphi \equiv \text{false} \mathcal{R} \varphi$, the *weak version of until* $\varphi \mathcal{W} \psi \equiv (\varphi \mathcal{U} \psi) \vee (\square \varphi)$, and *as soon as* $\varphi \mathcal{A} \psi \equiv \neg \psi \mathcal{W} (\psi \wedge \varphi)$.

Realizability. We are interested in the following realizability problem: given a TSL formula φ , is there a strategy $\sigma \in \mathcal{C}^{[\mathcal{I}^+]}$ such that for every input $\iota \in \mathcal{I}^\omega$ and function implementation $\langle \cdot \rangle : \mathbb{F} \rightarrow \mathcal{F}$, the branch $\sigma \wr \iota$ satisfies φ , i.e.,

$$\exists \sigma \in \mathcal{C}^{[\mathcal{I}^+]}. \forall \iota \in \mathcal{I}^\omega. \forall \langle \cdot \rangle : \mathbb{F} \rightarrow \mathcal{F}. \sigma \wr \iota, \iota \models_{\langle \cdot \rangle} \varphi$$

If such a strategy σ exists, we say σ realizes φ . If we additionally ask for a concrete instantiation of σ , we consider the synthesis problem of TSL.

5 TSL Properties

In order to synthesize programs from TSL specifications, we give an overview of the first part of our synthesis process, as shown in Fig. 1. First we show how to approximate the semantics of TSL through a reduction to LTL. However, due to the approximation, finding a realizable strategy immediately may fail. Our solution is a CEGAR loop that improves the approximation. This CEGAR loop is necessary, because the realizability problem of TSL is undecidable in general.

Approximating TSL with LTL. We approximate TSL formulas with weaker LTL formulas. The approximation reinterprets the syntactic elements, \mathcal{T}_P and \mathcal{T}_U , as atomic propositions for LTL. This strips away the semantic meaning of the function application and assignment in TSL, which we reconstruct by later adding assumptions lazily to the LTL formula.

Formally, let \mathcal{T}_P and \mathcal{T}_U be the finite sets of predicate terms and updates, which appear in φ_{TSL} , respectively. For every assigned signal, we partition \mathcal{T}_U into $\biguplus_{s_o \in \mathbb{O} \cup \mathbb{C}} \mathcal{T}_U^{s_o}$. For every $c \in \mathbb{C}$ let $\mathcal{T}_{U/\text{id}}^c = \mathcal{T}_U^c \cup \{\llbracket c \leftarrow c \rrbracket\}$, for $o \in \mathbb{O}$ let

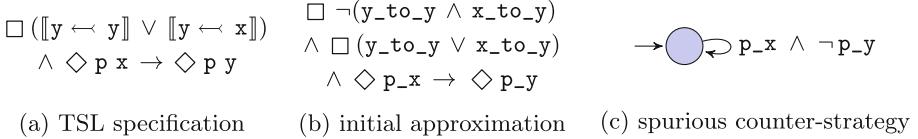


Fig. 5. A TSL specification (a) with input x and cell y that is realizable. A winning strategy is to save x to y as soon as $p(x)$ is satisfied. However, the initial approximation (b), that is passed to an LTL synthesis solver, is unrealizable, as proven through the counter-strategy (c) returned by the LTL solver.

$\mathcal{T}_{U/\text{id}}^o = \mathcal{T}_U^o$, and let $\mathcal{T}_{U/\text{id}} = \bigcup_{s_o \in \mathbb{O} \cup \mathbb{C}} \mathcal{T}_{U/\text{id}}^{s_o}$. We construct the LTL formula φ_{LTL} over the input propositions \mathcal{T}_F and output propositions $\mathcal{T}_{U/\text{id}}$ as follows:

$$\varphi_{LTL} = \square \left(\bigwedge_{s_o \in \mathbb{O} \cup \mathbb{C}} \bigvee_{\tau \in \mathcal{T}_{U/\text{id}}^{s_o}} (\tau \wedge \bigwedge_{\tau' \in \mathcal{T}_{U/\text{id}}^{s_o} \setminus \{\tau\}} \neg \tau') \right) \wedge \text{SYNTACTICCONVERSION}(\varphi_{TSL})$$

Intuitively, the first part of the equation partially reconstructs the semantic meaning of updates by ensuring that a signal is not updated with multiple values at a time. The second part extracts the reactive constraints of the TSL formula without the semantic meaning of functions and updates.

Theorem 1 ([19]). *If φ_{LTL} is realizable, then φ_{TSL} is realizable.*

Note that unrealizability of φ_{LTL} does not imply that φ_{TSL} is unrealizable. It may be that we have not added sufficiently many environment assumptions to the approximation in order for the system to produce a realizing strategy.

Example. As an example, we present a simple TSL specification in Fig. 5a. The specification asserts that the environment provides an input x for which the predicate $p x$ will be satisfied eventually. The system must guarantee that eventually $p y$ holds. According to the semantics of TSL the formula is realizable. The system can take the value of x when $p x$ is true and save it to y , thus guaranteeing that $p y$ is satisfied eventually. This is in contrast to LTL, which has no semantics for pure functions - taking the evaluation of $p y$ as an environmentally controlled value that does not need to obey the consistency of a pure function.

Refining the LTL Approximation. It is possible that the LTL solver returns a counter-strategy for the environment although the original TSL specification is realizable. We call such a counter-strategy *spurious* as it exploits the additional freedom of LTL to violate the purity of predicates as made possible by the underapproximation. Formally, a counter-strategy is an infinite tree $\pi: \mathcal{C}^* \rightarrow 2^{\mathcal{T}_F}$, which provides predicate evaluations in response to possible update assignments of function terms $\tau_F \in \mathcal{T}_F$ to outputs $o \in \mathbb{O}$. W.l.o.g. we can assume that \mathbb{O} , \mathcal{T}_F and \mathcal{T}_P are finite, as they can always be restricted to the outputs and terms that appear in the formula. A counter-strategy is spurious, iff there is a branch $\pi \wr \varsigma$ for some computation $\varsigma \in \mathcal{C}^\omega$, for which the strategy chooses an inconsistent evaluation of two equal predicate terms at different points in time, i.e.,

Algorithm 1. Check-Spuriousness

Input: bound b , counter-strategy $\pi: \mathcal{C}^* \rightarrow 2^{\mathcal{T}_P}$ (finitely represented using m states)

```

1: for all  $v \in \mathcal{C}^{m \cdot b}$ ,  $\tau_P \in \mathcal{T}_P$ ,  $t, t' \in \{0, 1, \dots, m \cdot b - 1\}$  do
2:   if  $\eta_{\langle\rangle_{\text{id}}}(v, \iota_{\text{id}}, t, \tau_P) \equiv \eta_{\langle\rangle_{\text{id}}}(v, \iota_{\text{id}}, t', \tau_P) \wedge$ 
       $\tau_P \in \pi(v_0 \dots v_{t-1}) \wedge \tau_P \notin \pi(v_0 \dots v_{t'-1})$  then
3:      $w \leftarrow \text{reduce}(v, \tau_P, t, t')$ 
4:     return  $\square(\bigwedge_{i=0}^{t-1} \bigcirc^i w_i \wedge \bigwedge_{i=0}^{t'-1} \bigcirc^i w_i \rightarrow (\bigcirc^t \tau_P \leftrightarrow \bigcirc^{t'} \tau_P))$ 
5: return “non-spurious”

```

$$\begin{aligned} & \exists \varsigma \in \mathcal{C}^\omega. \exists t, t' \in \mathbb{N}. \exists \tau_P \in \mathcal{T}_P. \\ & \quad \tau_P \in \pi(\varsigma(0)\varsigma(1) \dots \varsigma(t-1)) \wedge \tau_P \notin \pi(\varsigma(0)\varsigma(1) \dots \varsigma(t'-1)) \wedge \\ & \quad \forall \langle \cdot \rangle: \mathbb{F} \rightarrow \mathcal{F}. \eta_{\langle\rangle}(\varsigma, \pi \varsigma, t, \tau_P) = \eta_{\langle\rangle}(\varsigma, \pi \varsigma, t', \tau_P). \end{aligned}$$

Note that a non-spurious strategy can be inconsistent along multiple branches. Due to the definition of realizability the environment can choose function and predicate assignments differently against every system strategy accordingly.

By purity of predicates in TSL the environment is forced to always return the same value for predicate evaluations on equal values. However, this semantic property cannot be enforced implicitly in LTL. To resolve this issue we use the returned counter-strategy to identify spurious behavior in order to strengthen the LTL underapproximation with additional environment assumptions. After adding the derived assumptions, we re-execute the LTL synthesizer to check whether the added assumptions are sufficient in order to obtain a winning strategy for the system. If the solver still returns a spurious strategy, we continue the loop in a CEGAR fashion until the set of added assumptions is sufficiently complete. However, if a non-spurious strategy is returned, we have found a proof that the given TSL specification is indeed unrealizable and terminate.

Algorithm 1 shows how a returned counter-strategy π is checked for being spurious. To this end, it is sufficient to check π against system strategies bounded by the given bound b , as we use bounded synthesis [20]. Furthermore, we can assume w.l.o.g. that π is given by a finite state representation, which is always possible due to the finite model guarantees of LTL. Also note that π , as it is returned by the LTL synthesizer, responds to sequences of sets of updates $(2^{\mathcal{T}_{U/\text{id}}})^*$. However, in our case $(2^{\mathcal{T}_{U/\text{id}}})^*$ is an alternative representation of \mathcal{C}^* , due to the additional “single update” constraints added during the construction of φ_{LTL} .

The algorithm iterates over all possible responses $v \in \mathcal{C}^{m \cdot b}$ of the system up to depth $m \cdot b$. This is sufficient, since any deeper exploration would result in a state repetition of the cross-product of the finite state representation of π and any system strategy bounded by b . Hence, the same behaviour could also be generated by a sequence smaller than $m \cdot b$. At the same time, the algorithm iterates over predicates $\tau_P \in \mathcal{T}_P$ appearing in φ_{TSL} and times t and t' smaller than $m \cdot b$. For each of these elements, spuriousness is checked by comparing the output of π for the evaluation of τ_P at times t and t' , which should only differ if the inputs to the predicates are different as well. This can only happen, if the

passed input terms have been constructed differently over the past. We check it by using the evaluation function η equipped with the identity assignment $\langle \cdot \rangle_{\text{id}}: \mathbb{F} \rightarrow \mathbb{F}$, with $\langle f \rangle_{\text{id}} = f$ for all $f \in \mathbb{F}$, and the input sequence ι_{id} , with $\iota_{\text{id}}(t)(i) = (t, i)$ for all $t \in \mathbb{N}$ and $i \in \mathbb{I}$, that always generates a fresh input. Syntactic inequality of $\eta_{\langle \cdot \rangle_{\text{id}}}(v, \iota_{\text{id}}, t, \tau_P)$ and $\eta_{\langle \cdot \rangle_{\text{id}}}(v, \iota_{\text{id}}, t', \tau_P)$ then is a sufficient condition for the existence of an assignment $\langle \cdot \rangle: \mathbb{F} \rightarrow \mathcal{F}$, for which τ_P evaluates differently at times t and t' .

If spurious behaviour of π could be found, then the revealing response $v \in \mathcal{C}^*$ is first simplified using `reduce`, which reduces v again to a sequence of sets of updates $w \in (2^{\mathcal{T}_{U/\text{id}}})^*$ and removes updates that do not affect the behavior of τ_P at the times t and t' to accelerate the termination of the CEGAR loop. Afterwards, the sequence w is turned into a new assumption that prohibits the spurious behavior, generalized to prevent it even at arbitrary points in time.

As an example of this process, reconsider the spurious counter-strategy of Fig. 5c. Already after the first system response $\llbracket y \leftarrow x \rrbracket$, the environment produces an inconsistency by evaluating $p\ x$ and $p\ y$ differently. This is inconsistent, as the cell y holds the same value at time $t = 1$ as the input x at time $t = 0$. Using Algorithm 1 we generate the new assumption $\square(\llbracket y \leftarrow x \rrbracket \rightarrow (p\ x \leftrightarrow \bigcirc p\ y))$. After adding this strengthening the LTL synthesizer returns a realizability result.

Undecidability. Although we can approximate the semantics of TSL with LTL, there are TSL formulas that cannot be expressed as LTL formulas of finite size.

Theorem 2 ([19]). *The realizability problem of TSL is undecidable.*

6 TSL Synthesis

Our synthesis framework provides a modular refinement process to synthesize executables from TSL specifications, as depicted in Fig. 1. The user initially provides a TSL specification over predicate and function terms. At the end of the procedure, the user receives an executable to control a reactive system.

The first step of our method answers the synthesis question of TSL: if the specification is realizable, then a control flow model is returned. To this end, an intermediate translation to LTL is used, utilizing an LTL synthesis solver that produces circuits in the AIGER format. If the specification is realizable, the resulting control flow model is turned into Haskell code, which is implemented as an independent Haskell module. The user has the choice between two different targets: a module built on Arrows, which is compatible with any Arrowized FRP library, or a module built on Applicative, which supports Applicative FRP libraries. Our procedure generates a single Haskell module per TSL specification. This makes naturally decomposing a project according to individual tasks possible. Each module provides a single component, which is parameterized by their initial state and the pure function and predicate transformations. As soon as these are provided as part of the surrounding project context, a final executable can be generated by compiling the Haskell code.

An important feature of our synthesis approach is that implementations for the terms used in the specification are only required after synthesis. This allows

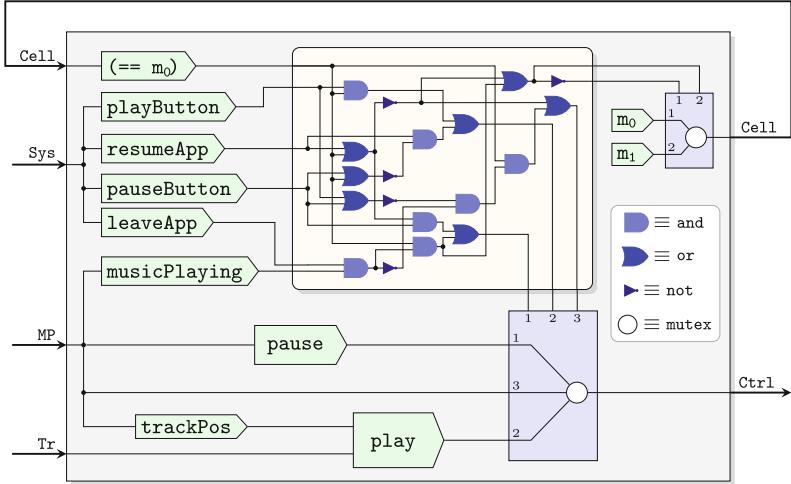


Fig. 6. Example CFM of the music player generated from a TSL specification.

the user to explore several possible specifications before deciding on any term implementations.

Control Flow Model. The first step of our approach is the synthesis of a *Control Flow Model* \mathcal{M} (CFM) from the given TSL specification φ , which provides us with a uniform representation of the control flow structure of our final program.

Formally, a CFM \mathcal{M} is a tuple $\mathcal{M} = (\mathbb{I}, \mathbb{O}, \mathbb{C}, V, \ell, \delta)$, where \mathbb{I} is a finite set of inputs, \mathbb{O} is a finite set of outputs, \mathbb{C} is a finite set of cells, V is a finite set of vertices, $\ell: V \rightarrow \mathbb{F}$ assigns a vertex a function $f \in \mathbb{F}$ or a predicate $p \in \mathbb{P}$, and

$$\delta: (\mathbb{O} \cup \mathbb{C} \cup V) \times \mathbb{N} \rightarrow (\mathbb{I} \cup \mathbb{C} \cup V \cup \{\perp\})$$

is a dependency relation that relates every output, cell, and vertex of the CFM with $n \in \mathbb{N}$ arguments, which are either inputs, cells, or vertices. Outputs and cells $s \in \mathbb{O} \cup \mathbb{C}$ always have only a single argument, i.e., $\delta(s, 0) \neq \perp$ and $\forall m > 0. \delta(s, m) \equiv \perp$, while for vertices $x \in V$ the number of arguments $n \in \mathbb{N}$ align with the arity of the assigned function or predicate $\ell(x)$, i.e., $\forall m \in \mathbb{N}. \delta(x, m) \equiv \perp \leftrightarrow m > n$. A CFM is valid if it does not contain circular dependencies, i.e., on every cycle induced by δ there must lie at least a single cell. We only consider valid CFMs.

An example CFM for our music player of Sect. 2 is depicted in Fig. 6. Inputs \mathbb{I} come from the left and outputs \mathbb{O} leave on the right. The example contains a single cell $c \in \mathbb{C}$, which holds the stateful memory `Cell`, introduced during synthesis for the module. The green, arrow shaped boxes depict vertices V , which are labeled with functions and predicates names, according to ℓ . For the Boolean decisions that define δ , we use circuit symbols for conjunction, disjunction, and negation. Boolean decisions are piped to a multiplexer gate that selects the respective update stream to be passed to an

output stream if and only if the respective Boolean trigger evaluates positively, while our construction ensures mutual exclusion on the Boolean triggers. For code generation, the logic gates are implemented using the corresponding dedicated Boolean functions. After building a control structure, we assign semantics to functions and predicates by providing implementations. To this end, we use Functional Reactive Programming (FRP). Prior work has established Causal Commutative Arrows (CCA) as an FRP language pattern equivalent to a CFM [33, 34, 53]. CCAs are an abstraction subsumed by other functional reactive programming abstractions, such as Monads, Applicative and Arrows [32, 33]. There are many FRP libraries using Monads [11, 14, 42], Applicative [2, 3, 23, 48], or Arrows [10, 39, 41, 51], and since every Monad is also an Applicative and Applicative/Arrows both are universal design patterns, we can give uniform translations to all of these libraries using translations to just Applicative and Arrows. Both translations are possible due to the flexible notion of a CFM.

In the last step, the synthesized FRP program is compiled into an executable, using the provided function and predicate implementations. This step is not fixed to a single compiler implementation, but in fact can use any FRP compiler (or library) that supports a language abstraction at least as expressive as CCA. For example, instead of creating an Android music player app, we could target an FRP web interface [48] to create an online music player, or an embedded FRP library [23] to instantiate the player on a computationally more restricted device. By using the strong core of CCA, we even can directly implement the player in hardware, which is for example possible with the CλaSH compiler [3]. Note that we still need separate implementations for functions and predicates for each target. However, the specification and synthesized CFM always stay the same.

7 Experimental Results

To evaluate our synthesis procedure we implemented a tool that follows the structure of Fig. 1. It first encodes the given TSL specification in LTL and then refines it until an LTL solver either produces a realizability result or returns a non-spurious counter-strategy. For LTL synthesis we use the bounded synthesis tool BoSy [15]. As soon as we get a realizing strategy it is translated to a corresponding CFM. Then, we generate the FRP program structure. Finally, after providing function implementations the result is compiled into an executable.

To demonstrate the effectiveness of synthesizing TSL, we applied our tool to a collection of benchmarks from different application domains, listed in Table 1. Every benchmark class consists of multiple specifications, addressing different features of TSL. We created all specifications from scratch, where we took care that they either relate to existing textual specifications, or real world scenarios. A short description of each benchmark class is given in [19].

For every benchmark, we report the synthesis time and the size of the synthesized CFM, split into the number of cells ($|C_M|$) and vertices ($|V_M|$) used. The synthesized CFM may use more cells than the original TSL specification if synthesis requires more memory in order to realize a correct control flow.

Table 1. Number of cells $|C_M|$ and vertices $|V_M|$ of the resulting CFM M and synthesis times for a collection of TSL specifications φ . A * indicates that the benchmark additionally has an initial condition as part of the specification.

BENCHMARK (φ)	$ \varphi $	$ \mathbb{I} $	$ \mathbb{O} $	$ \mathbb{P} $	$ \mathbb{F} $	$ C_M $	$ V_M $	SYNTHESIS TIME (s)
Button default	7	1	2	1	3	3	8	0.364
Music App simple	91	3	1	4	7	2	25	0.77
system feedback	103	3	1	5	8	2	31	0.572
motivating example	87	3	1	5	8	2	70	1.783
FRPZoo scenario ₀	54	1	3	2	8	4	36	1.876
scenario ₅	50	1	3	2	7	4	32	1.196
scenario ₁₀	48	1	3	2	7	4	32	1.161
Escalator non-reactive	8	0	1	0	1	2	4	0.370
non-counting	15	2	1	2	4	2	19	0.304
counting	34	2	2	3	7	3	23	0.527
counting*	43	2	2	3	8	4	43	0.621
bidirectional	111	2	2	5	10	3	214	4.555
bidirectional*	124	2	2	5	11	4	287	16.213
smart	45	2	1	2	4	4	159	24.016
Slider default	50	1	1	2	4	2	15	0.664
scored	67	1	3	4	8	4	62	3.965
delayed	71	1	3	4	8	5	159	7.194
Haskell-TORCS simple	40	5	3	2	16	4	37	0.680
advanced gearing	23	4	1	1	3	2	7	0.403
accelerating	15	2	2	2	6	3	11	0.391
steering simple	45	2	1	4	6	2	31	0.459
improved	100	2	2	4	10	3	26	1.347
smart	76	3	2	4	8	5	227	3.375

Table 2. Set of programs that use purity to keep one or two counters in range. Synthesis needs multiple refinements of the specification to proof realizability.

BENCHMARK (φ)	$ \varphi $	$ \mathbb{I} $	$ \mathbb{O} $	$ \mathbb{P} $	$ \mathbb{F} $	$ C_M $	$ V_M $	REFINEMENTS	SYNTHESIS TIME (s)
inrange-single	23	2	1	2	4	2	21	3	0.690
inrange-two	51	3	3	4	7	4	440	6	173.132
graphical-single	55	2	3	2	6	4	343	9	1767.948
graphical-two	113	3	5	4	9	-	-	-	$\zeta 10000$

The synthesis was executed on a quad-core Intel Xeon processor (E3-1271 v3, 3.6GHz, 32 GB RAM, PC1600, ECC), running Ubuntu 64bit LTS 16.04.

The experiments of Table 1 show that TSL successfully lifts the applicability of synthesis from the Boolean domain to arbitrary data domains, allowing for new applications that utilize every level of required abstraction. For all benchmarks we always found a realizable system within a reasonable amount of time, where the results often required synthesized cells to realize the control flow behavior.

We also considered a preliminary set of benchmarks that require multiple refinement steps to be synthesizable. An overview of the results is given in Table 2. The benchmarks are inspired by examples of the Reactive Banana FRP library [2]. Here, purity of function and predicate applications must be utilized by the system to ensure that the value of one or two counters never goes out of range. Thereby, the system not only needs purity to verify this condition, but also to take the correct decisions in the resulting implementation to be synthesized.

8 Related Work

Our approach builds on the rich body of work on reactive synthesis, see [17] for a survey. The classic reactive synthesis problem is the construction of a finite-state machine that satisfies a specification in a temporal logic like LTL. Our approach differs from the classic problem in its connection to an actual programming paradigm, namely FRP, and its separation of control and data.

The synthesis of *reactive programs*, rather than finite-state machines, has previously been studied for standard temporal logic [21, 35]. Because there is no separation of control and data, these approaches do not directly scale to realistic applications. With regard to FRP, a *Curry-Howard correspondence* between LTL and FRP in a dependently typed language was discovered [28, 29] and used to prove properties of FRP programs [8, 30]. However, our paper is the first, to the best of our knowledge, to study the synthesis of FRP programs from temporal specifications.

The idea to separate control and data has appeared, on a smaller scale, in the synthesis with *identifiers*, where identifiers, such as the number of a client in a mutual exclusion protocol, are treated symbolically [13]. *Uninterpreted functions* have been used to abstract data-related computational details in the synthesis of synchronization primitives for complex programs [5]. Another connection to other synthesis approaches is our CEGAR loop. Similar *refinement loops* also appear in other synthesis approaches, however with a different purpose, such as the refinement of environment assumptions [1].

So far, there is no immediate connection between our approach and the substantial work on *deductive* and *inductive synthesis*, which is specifically concerned with the data-transformation aspects of programs [16, 31, 40, 47, 49, 50]. Typically, these approaches are focussed on non-reactive sequential programs. An integration of deductive and inductive techniques into our approach for reactive systems is a very promising direction for future work. Abstraction-based synthesis [4, 12, 24, 37] may potentially provide a link between the approaches.

9 Conclusions

We have introduced Temporal Stream Logic, which allows the user to specify the control flow of a reactive program. The logic cleanly separates control from complex data, forming the foundation for our procedure to synthesize FRP programs. By utilizing the purity of function transformations our logic scales independently of the complexity of the data to be handled. While we have shown that scalability comes at the cost of undecidability, we addressed this issue by using a CEGAR loop, which lazily refines the underapproximation until either a realizing system implementation or an unrealizability proof is found.

Our experiments indicate that TSL synthesis works well in practice and on a wide range of programming applications. TSL also provides the foundations for further extensions. For example, a user may want to fix the semantics for a subset of the functions and predicates. Such refinements can be implemented as part of a much richer *TSL Modulo Theory* framework.

References

1. Alur, R., Moarref, S., Topcu, U.: Counter-strategy guided refinement of GR(1) temporal logic specifications. In: Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, 20–23 October 2013, pp. 26–33. IEEE (2013). <http://ieeexplore.ieee.org/document/6679387/>
2. Apfelmus, H.: Reactive-banana. Haskell library (2012). <http://www.haskell.org/haskellwiki/Reactive-banana>
3. Baaij, C.: Digital circuit in CλaSH: functional specifications and type-directed synthesis. Ph.D. thesis, University of Twente, January 2015. <https://doi.org/10.3990/1.9789036538039>, eemcs-eprint-23939
4. Beyene, T.A., Chaudhuri, S., Popescu, C., Rybalchenko, A.: A constraint-based approach to solving games on infinite graphs. In: Jagannathan and Sewell [26], pp. 221–234. <https://doi.org/10.1145/2535838.2535860>, <http://doi.acm.org/10.1145/2535838.2535860>
5. Bloem, R., Hofferek, G., Könighofer, B., Könighofer, R., Ausserlechner, S., Spork, R.: Synthesis of synchronization using uninterpreted functions. In: Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, 21–24 October 2014, pp. 35–42. IEEE (2014). <https://doi.org/10.1109/FMCAD.2014.6987593>
6. Bloem, R., Jacobs, S., Khalimov, A.: Parameterized synthesis case study: AMBA AHb. In: Chatterjee, K., Ehlers, R., Jha, S. (eds.) Proceedings 3rd Workshop on Synthesis, SYNT 2014. EPTCS, Vienna, Austria, 23–24 July 2014, vol. 157, pp. 68–83 (2014). <https://doi.org/10.4204/EPTCS.157.9>
7. 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). <https://doi.org/10.1016/j.jcss.2011.08.007>
8. Cave, A., Ferreira, F., Panangaden, P., Pientka, B.: Fair reactive programming. In: Jagannathan and Sewell [26], pp. 361–372. <https://doi.org/10.1145/2535838.2535881>, <http://doi.acm.org/10.1145/2535838.2535881>
9. 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). <https://doi.org/10.1145/876638.876643>

10. Courtney, A., Nilsson, H., Peterson, J.: The yampa arcade. In: Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2003, Uppsala, Sweden, 28 August 2003, pp. 7–18. ACM, (2003). <https://doi.org/10.1145/871895.871897>, <http://doi.acm.org/10.1145/871895.871897>
11. Czaplicki, E., Chong, S.: Asynchronous functional reactive programming for Guis. In: Boehm, H., Flanagan, C. (eds.) ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2013, Seattle, WA, USA, 16–19 June 2013, pp. 411–422. ACM (2013). <https://dl.acm.org/citation.cfm?doid=2462156.2462161>, <http://doi.acm.org/10.1145/2462156.2462161>
12. Dimitrova, R., Finkbeiner, B.: Counterexample-guided synthesis of observation predicates. In: Jurdziński, M., Ničković, D. (eds.) FORMATS 2012. LNCS, vol. 7595, pp. 107–122. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33365-1_9
13. Ehlers, R., Seshia, S.A., Kress-Gazit, H.: Synthesis with identifiers. In: McMillan, K.L., Rival, X. (eds.) VMCAI 2014. LNCS, vol. 8318, pp. 415–433. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54013-4_23
14. Elliott, C., Hudak, P.: Functional reactive animation. In: Jones, S.L.P., Tofte, M., Berman, A.M. (eds.) Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP 1997), Amsterdam, The Netherlands, 9–11 June 1997, pp. 263–273. ACM (1997). <https://doi.org/10.1145/258948.258973>, <http://doi.acm.org/10.1145/258948.258973>
15. Faymonville, P., Finkbeiner, B., Tentrup, L.: BoSy: an experimentation framework for bounded synthesis. In: Majumdar, R., Kunčák, V. (eds.) CAV 2017. LNCS, vol. 10427, pp. 325–332. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63390-9_17
16. Feser, J.K., Chaudhuri, S., Dillig, I.: Synthesizing data structure transformations from input-output examples. In: Grove and Blackburn [22], pp. 229–239. <https://doi.org/10.1145/2737924.2737977>, <http://doi.acm.org/10.1145/2737924.2737977>
17. Finkbeiner, B.: Synthesis of reactive systems. In: Esparza, J., Grumberg, O., Sickert, S. (eds.) Dependable Software Systems Engineering. NATO Science for Peace and Security Series, D: Information and Communication Security, vol. 45, pp. 72–98. IOS Press (2016)
18. Finkbeiner, B., Klein, F., Piskac, R., Santolucito, M.: Vehicle platooning simulations with functional reactive programming. In: Proceedings of the 1st International Workshop on Safe Control of Connected and Autonomous Vehicles, SCAV@CPSWeek 2017, Pittsburgh, PA, USA, 21 April 2017, pp. 43–47. ACM, (2017). <https://doi.org/10.1145/3055378.3055385>, <http://doi.acm.org/10.1145/3055378.3055385>
19. Finkbeiner, B., Klein, F., Piskac, R., Santolucito, M.: Temporal stream logic: Synthesis beyond the bools. CoRR abs/1712.00246 (2019). <http://arxiv.org/abs/1712.00246>
20. Finkbeiner, B., Schewe, S.: Bounded synthesis. STTT **15**(5–6), 519–539 (2013). <https://doi.org/10.1007/s10009-012-0228-z>
21. Gerstacker, C., Klein, F., Finkbeiner, B.: Bounded synthesis of reactive programs. In: Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, 7–10 October 2018, Proceedings, pp. 441–457 (2018). https://doi.org/10.1007/978-3-030-01090-4_26
22. Grove, D., Blackburn, S. (eds.): Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, 15–17 June 2015. ACM (2015). <http://dl.acm.org/citation.cfm?id=2737924>

23. Helbling, C., Guyer, S.Z.: Juniper: a functional reactive programming language for the arduino. In: Janin and Sperber [27], pp. 8–16. <https://doi.org/10.1145/2975980.2975982>, <http://doi.acm.org/10.1145/2975980.2975982>
24. Hsu, K., Majumdar, R., Mallik, K., Schmuck, A.K.: Multi-layered abstraction-based controller synthesis for continuous-time systems. In: Proceedings of the 21st International Conference on Hybrid Systems: Computation and Control (part of CPS Week), pp. 120–129. ACM (2018)
25. Jacobs, S., et al.: The 4th reactive synthesis competition (SYNTCOMP 2017): Benchmarks, participants and results. In: SYNT 2017. EPTCS, vol. 260, pp. 116–143 (2017). <https://doi.org/10.4204/EPTCS.260.10>
26. Jagannathan, S., Sewell, P. (eds.): The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2014, San Diego, CA, USA, 20–21 January 2014. ACM (2014). <http://dl.acm.org/citation.cfm?id=2535838>
27. Janin, D., Sperber, M. (eds.): Proceedings of the 4th International Workshop on Functional Art, Music, Modelling, and Design, FARM@ICFP 2016, Nara, Japan, 24 September 2016. ACM (2016). <https://doi.org/10.1145/2975980>, <http://doi.acm.org/10.1145/2975980>
28. Jeffrey, A.: LTL types FRP: linear-time temporal logic propositions as types, proofs as functional reactive programs. In: Claessen, K., Swamy, N. (eds.) Proceedings of the sixth workshop on Programming Languages meets Program Verification, PLPV 2012, Philadelphia, PA, USA, 24 January 2012, pp. 49–60. ACM (2012). <https://doi.org/10.1145/2103776.2103783>, <http://doi.acm.org/10.1145/2103776.2103783>
29. Jeltsch, W.: Towards a common categorical semantics for linear-time temporal logic and functional reactive programming. Electr. Notes Theor. Comput. Sci. **286**, 229–242 (2012). <https://doi.org/10.1016/j.entcs.2012.08.015>
30. Krishnaswami, N.R.: Higher-order functional reactive programming without space-time leaks. In: Morrisett, G., Uustalu, T. (eds.) ACM SIGPLAN International Conference on Functional Programming, ICFP 2013, Boston, MA, USA, 25–27 September 2013, pp. 221–232. ACM (2013). <https://doi.org/10.1145/2500365.2500588>, <http://doi.acm.org/10.1145/2500365.2500588>
31. Kuncak, V., Mayer, M., Piskac, R., Suter, P.: Comfy: a tool for complete functional synthesis. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 430–433. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_38
32. Lindley, S., Wadler, P., Yallop, J.: Idioms are oblivious, arrows are meticulous, monads are promiscuous. Electr. Notes Theor. Comput. Sci. **229**(5), 97–117 (2011). <https://doi.org/10.1016/j.entcs.2011.02.018>
33. Liu, H., Cheng, E., Hudak, P.: Causal commutative arrows. J. Funct. Program. **21**(4–5), 467–496 (2011). <https://doi.org/10.1017/S0956796811000153>
34. Liu, H., Hudak, P.: Plugging a space leak with an arrow. Electr. Notes Theor. Comput. Sci. **193**, 29–45 (2007). <https://doi.org/10.1016/j.entcs.2007.10.006>
35. Madhusudan, P.: Synthesizing reactive programs. In: Bezem, M. (ed.) Computer Science Logic, 25th International Workshop/20th Annual Conference of the EACSL, CSL 2011, Bergen, Norway, 12–15 September 2011, Proceedings. LIPIcs, vol. 12, pp. 428–442. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2011). <https://doi.org/10.4230/LIPIcs.CSL.2011.428>
36. Mainland, G. (ed.): Proceedings of the 9th International Symposium on Haskell, Haskell 2016, Nara, Japan, 22–23 September 2016. ACM (2016). <https://doi.org/10.1145/2976002>, <http://doi.acm.org/10.1145/2976002>

37. Mallik, K., Schmuck, A.K., Soudjani, S., Majumdar, R.: Compositional abstraction-based controller synthesis for continuous-time systems. arXiv preprint [arXiv:1612.08515](https://arxiv.org/abs/1612.08515) (2016)
38. Manna, Z., Waldinger, R.: A deductive approach to program synthesis. ACM Trans. Program. Lang. Syst. **2**(1), 90–121 (1980). <https://doi.org/10.1145/357084.357090>
39. Murphy, T.E.: A livecoding semantics for functional reactive programming. In: Janin and Sperber [27], pp. 48–53. <https://doi.org/10.1145/2975980.2975986> <http://doi.acm.org/10.1145/2975980.2975986>
40. Osera, P., Zdancewic, S.: Type-and-example-directed program synthesis. In: Grove and Blackburn [22], pp. 619–630. <https://doi.org/10.1145/2737924.2738007>, <http://doi.acm.org/10.1145/2737924.2738007>
41. Perez, I., Bärenz, M., Nilsson, H.: Functional reactive programming, refactored. In: Mainland [36], pp. 33–44. <https://doi.org/10.1145/2976002.2976010>, <http://doi.acm.org/10.1145/2976002.2976010>
42. van der Ploeg, A., Claessen, K.: Practical principled FRP: forget the past, change the future, FRPNow! In: Fisher, K., Reppy, J.H. (eds.) Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, 1–3 September 2015, pp. 302–314. ACM (2015). <https://doi.org/10.1145/2784731.2784752>, <http://doi.acm.org/10.1145/2784731.2784752>
43. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October–1 November 1977, pp. 46–57. IEEE Computer Society (1977). <https://doi.org/10.1109/SFCS.1977.32>
44. Pnueli, A., Rosner, R.: On the synthesis of an asynchronous reactive module. In: Ausiello, G., Dezani-Ciancaglini, M., Della Rocca, S.R. (eds.) ICALP 1989. LNCS, vol. 372, pp. 652–671. Springer, Heidelberg (1989). <https://doi.org/10.1007/BFb0035790>
45. Post, E.L.: A variant of a recursively unsolvable problem. Bull. Am. Math. Soc. **52**(4), 264–268 (1946). <http://projecteuclid.org/euclid.bams/1183507843>
46. Shan, Z., Azim, T., Neamtiu, I.: Finding resume and restart errors in android applications. In: Visser, E., Smaragdakis, Y. (eds.) Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, 30 October–4 November 2016, pp. 864–880. ACM (2016). <https://doi.org/10.1145/2983990.2984011>, <http://doi.acm.org/10.1145/2983990.2984011>
47. Solar-Lezama, A.: Program sketching. STTT **15**(5–6), 475–495 (2013). <https://doi.org/10.1007/s10009-012-0249-7>
48. Trinkle, R.: Reflex-frp (2017). <https://github.com/reflex-frp/reflex>
49. Vechev, M.T., Yahav, E., Yorsh, G.: Abstraction-guided synthesis of synchronization. STTT **15**(5–6), 413–431 (2013). <https://doi.org/10.1007/s10009-012-0232-3>
50. Wang, X., Dillig, I., Singh, R.: Synthesis of data completion scripts using finite tree automata. PACMPL 1(OOPSLA), 62:1–62:26 (2017). <https://doi.org/10.1145/3133886>, <http://doi.acm.org/10.1145/3133886>
51. Winograd-Cort, D.: Effects, Asynchrony, and Choice in Arrowized Functional Reactive Programming. Ph.D. thesis, Yale University, December 2015. <http://www.danwc.com/s/dwc-yale-formatted-dissertation.pdf>
52. Wongpiromsarn, T., Topcu, U., Murray, R.M.: Synthesis of control protocols for autonomous systems. Unmanned Syst. **1**(01), 21–39 (2013)

53. Yallop, J., Liu, H.: Causal commutative arrows revisited. In: Mainland [36], pp. 21–32. <https://doi.org/10.1145/2976002.2976019>, <http://doi.acm.org/10.1145/2976002.2976019>

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Run-Time Optimization for Learned Controllers Through Quantitative Games

Guy Avni¹⁽⁾, Roderick Bloem², Krishnendu Chatterjee¹, Thomas A. Henzinger¹,
Bettina Könighofer², and Stefan Pranger²

¹ IST Austria, Klosterneuburg, Austria

guy.avni@ist.ac.at

² TU Graz, Graz, Austria

Abstract. A controller is a device that interacts with a plant. At each time point, it reads the plant’s state and issues commands with the goal that the plant operates optimally. Constructing optimal controllers is a fundamental and challenging problem. Machine learning techniques have recently been successfully applied to train controllers, yet they have limitations. Learned controllers are monolithic and hard to reason about. In particular, it is difficult to add features without retraining, to guarantee any level of performance, and to achieve acceptable performance when encountering untrained scenarios. These limitations can be addressed by deploying quantitative run-time *shields* that serve as a proxy for the controller. At each time point, the shield reads the command issued by the controller and may choose to alter it before passing it on to the plant. We show how optimal shields that interfere as little as possible while guaranteeing a desired level of controller performance, can be generated systematically and automatically using reactive synthesis. First, we abstract the plant by building a stochastic model. Second, we consider the learned controller to be a black box. Third, we measure *controller performance* and *shield interference* by two quantitative run-time measures that are formally defined using weighted automata. Then, the problem of constructing a shield that guarantees maximal performance with minimal interference is the problem of finding an optimal strategy in a stochastic 2-player game “controller versus shield” played on the abstract state space of the plant with a quantitative objective obtained from combining the performance and interference measures. We illustrate the effectiveness of our approach by automatically constructing lightweight shields for learned traffic-light controllers in various road networks. The shields we generate avoid liveness bugs, improve controller performance in untrained and changing traffic situations, and add features to learned controllers, such as giving priority to emergency vehicles.

1 Introduction

The *controller synthesis* problem is a fundamental problem that is widely studied by different communities [42, 44]. A controller is a device that interacts with a *plant*. In each point in time it reads the plant’s state, e.g., given by sensor reading, and issues

This research was supported in part by the Austrian Science Fund (FWF) under grants S114 (RiSE/SHiNE), Z211-N23 (Wittgenstein Award), and M 2369-N33 (Meitner fellowship).

© The Author(s) 2019

I. Dillig and S. Tasiran (Eds.): CAV 2019, LNCS 11561, pp. 630–649, 2019.

https://doi.org/10.1007/978-3-030-25540-4_36

a command based on the state. The controller should guarantee that the plant operates correctly or optimally with respect to some given specification. As a running example, we consider a traffic light controller for a road intersection (see Fig. 1). The state of the plant refers to the state of the roads leading to the junction; namely, the positions of the cars, their speeds, their sizes, etc. A controller command consists of a light configuration for the junction in the next time frame. Specifications can either be qualitative, e.g., “it should never be the case that a road with an empty queue gets a green light”, or quantitative, e.g., “the cost of a controller is the average waiting times of the cars in the junction”.

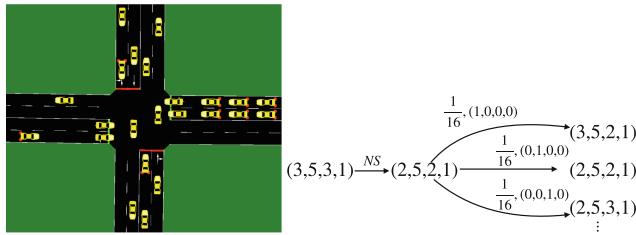


Fig. 1. On the left, a concrete state depicted in the traffic simulator SUMO. On the right, we depict the corresponding abstract state with queues cut off at $k = 5$, and some outgoing transitions. Upon issuing action North-South, a car is evicted from each of the North-South queues. Then, we choose uniformly at random, out of the 16 possible options, the incoming cars to the queues, update the state, and cutoff the queues at k (e.g., when a car enters from East, the queue stays 5).

A challenge in controller synthesis is that, since the number of possible plant readings is huge, it is computationally demanding to find an optimal command, given a plant state. Machine learning is a prominent approach to make decisions based on large amounts of collected data [28, 37]. It is widely successful in practice and takes an integral part in the design process of various systems. Machine learning has been successfully applied to train controllers [15, 33, 34] and specifically controllers for traffic control [20, 35, 39].

A shortcoming of machine-learning techniques is that the controllers that are produced are black-box devices that are hard to reason about and modify without a complete re-training. It is thus challenging, for example, to obtain worst-case guarantees about the controller, which is particularly important in safety-critical settings. Attempts to address this problem come from both the formal methods community [46], where verification of learned systems is extensively studied [24, 29], and the machine-learning community, where guarantees are added during the training process using reward engineering [13, 18] or by modifying the exploration process [11, 19, 38]. Both approaches require expertise in the respective field and suffer from limitations such as scalability for the first, and intricacy and robustness issues, for the second. Moreover, both techniques were mostly studied for safety properties.

Another shortcoming of machine-learning techniques is that they require expertise and a fine-tuning of parameters. It is difficult, for example, to train controllers that are

robust to plant behaviors, e.g., a controller that has been trained on uniform traffic congestion meeting rush-hour traffic, which can be significantly different and can cause poor performance. Also, it is challenging to add features to a controller without retraining, which is both costly and time consuming. These can include permanent features, e.g., priority to public transport, or temporary changes, e.g., changes due to an accident or construction. Again, since the training process is intricate, adding features during training can have unexpected effects.

In this work, we use quantitative *shields* to deal with the limitations of learned or any other black-box controllers. A shield serves as a proxy between the controller and the plant. In each point in time, as before, the controller reads the state of the plant and issues a command. Rather than directly feeding the command to the plant, the shield first reads it along with an abstract plant state. The shield can then choose to keep the controller's command or alter it, before issuing the command to the plant. The concept of shields was first introduced in [30], where shields for safety properties were considered and with a qualitative notion of interference: a shield is only allowed to interfere when a controller error occurs, which is only well-defined when considering safety properties. We elaborate on other shield-like approaches in the Sect. 1.1.

Our goal is to automatically synthesize shields that optimize quantitative measures for black-box controllers. We are interested in synthesizing lightweight shields. We assume that the controller performs well on average, but has no worst-case guarantees. When combining the shield and the controller, intuitively, the controller should be active for the majority of the time and the shield intervenes only when it is required. We formalize the plant behavior as well as the interference cost using quantitative measures. Unlike safety objectives, where it is clear when a shield must interfere, with quantitative objectives, a non-interference typically does not have a devastating effect. It is thus challenging to decide, at each time point, whether the shield should interfere or not; the shield needs to balance the cost of interfering with the decrease in performance of not interfering. Automatic synthesis of shields is thus natural in this setting.

We elaborate on the two quantitative measures we define. The interaction between the plant, controller, and shield gives rise to an infinite sequence over $C \times \Gamma \times \Gamma$, where C is a set of plant states and Γ is a set of allowed actions. A triple $\langle c, \gamma_1, \gamma_2 \rangle$ means that the plant is in state c , the controller issues command γ_1 , and the shield (possibly) alters it to γ_2 . We use *weighted automata* to assign costs to infinite traces, which have proven to be a convenient, flexible, and robust quantitative specification language [14]. Our *behavioral score* measures the performance of the plant and it is formally given by a weighted automaton that assigns scores to traces over $C \times \Gamma$. Boolean properties are a special case, which include *safety* properties, e.g., “an emergency vehicle should always get a green light”, and *liveness*, e.g., “a car waiting in a queue eventually gets the green light”. An example of a quantitative score is the long-run average of the waiting times of the vehicles in the city. A second score measures the *interference* of a shield with a controller. It is given by a weighted automaton over the alphabet $\Gamma \times \Gamma$. A simple example of an interference score charges the shield 1 for every change of action and charges 0 when no change is made. Then, the score of an infinite trace can be phrased as the ratio of the time that the shield interferes. Using weighted automata we can specify more involved scores such as different charges for different types of alterations or even

charges that depend on the past, e.g., altering the controller’s command twice in a row is not allowed.

Given a probabilistic plant model and a formal specification of behavioral and interference scores, the problem of synthesizing an optimal shield is well-defined and can be solved by game theory. While the game-based techniques we use are those of discrete-event controller synthesis [3] in a stochastic setting with quantitative objectives, our set-up is quite different. In traditional controller synthesis, there are two entities; the controller and the adversarial plant. The goal is to synthesize a controller offline. In our setting, there are three entities: the plant, whose behavior we model probabilistically, the controller, which we treat as a black-box and model as an adversary, and the shield, which we synthesize. Note that the shield’s synthesis procedure is done offline but it makes online decisions when it operates together with the controller and plant. Our plant model is formally given by a *Markov decision process* which is a standard model with which one models lack of knowledge about the plant using probability (see Fig. 1 and details in Example 1). The game is played on the MDP by two players; a shield and a controller, where the quantitative objective is given by the two scores. An optimal shield is then extracted from an optimal strategy for the shield player. The game we construct admits memoryless optimal strategies, thus the size of the shield is proportional to the size of the abstraction of the plant. In addition, it is implemented as a look-up table for actions in every state. Thus, the runtime overhead is a table look-up and hence negligible.

We experiment with our framework by constructing shields for traffic lights in a network of roads. Our experimental results illustrate the usefulness of the framework. We construct shields that consistently improve the performance of controllers, especially when exhibiting behavior that they are not trained on, but, more surprising, also while exhibiting trained behavior. We show that the use of a shield reduces variability in performance among various controllers, thus when using a shield, the choice of the parameters used in the training phase becomes less acute. We show how a shield can be used to add the functionality of prioritizing public transport as well as local fairness to a controller, both without re-training the controller. In addition, we illustrate how shields can add worst-case guarantees on liveness without a costly verification of the controller.

1.1 Related Work

A shield-like approach to adding safety to systems is called *runtime assurance* [47], and has applications, for example, in control of robotics [41] and drones [12]. In this framework, a switching mechanism alternates between running a high-performance system and a provably safe one. These works differ from ours since they consider safety specifications. As mentioned earlier, a challenge with quantitative specifications is that, unlike safety specifications, a non-interference typically does not have a devastating effect, thus it is not trivial to decide when and to what extent to interfere.

Another line of work is *runtime enforcement*, where an enforcer monitors a program that outputs events and can either terminate the program once it detects an error [45], or alter the event in order to guarantee, for example, safety [21], richer qualitative objectives [16], or privacy [26,49]. The similarities between an enforcer and a shield is in

their ability to alter events. The settings are quite different, however, since the enforced program is not reactive whereas we consider a plant that receives commands.

Recently, formal approaches were proposed in order to restrict the exploration of the learning agent such that a set of logically constraints are always satisfied. This method can support other properties beyond safety, e.g., probabilistic computation tree logic (PCTL) [25, 36], linear temporal logic (LTL) [1], or differential dynamic logic [17]. To the best of our knowledge, quantitative specifications have not yet been considered. Unlike these approaches, we consider the learned controller as a black box, thus our approach is particularly suitable for machine learning non-experts.

While MDPs and partially-observable MDPs have been widely studied in the literature w.r.t. to quantitative objectives [27, 43], our framework requires the interaction of two players (the shield and the black-box controller) and we use game-theoretic framework with quantitative objectives for our solution.

2 Definitions and Problem Statement

2.1 Plants, Controllers, and Shields

The interaction with a *plant* over a concrete set of states C is carried out using two functionalities: PLANT.GETSTATE returns the plant’s current state and PLANT.ISSUECOMMAND issues an action from a set Γ . Once an action is issued, the plant updates its state according to some unknown transition function. At each point in time, the *controller* reads the state of the plant and issues a command. Thus, it is a function from a history in $(C \times \Gamma)^* \cdot C$ to Γ .

Informally, a *shield* serves as a proxy between the controller and the plant. In each time point, it reads the controller’s issued action and can choose an alternative action to issue to the plant. We are interested in light-weight shields that add little or no overhead to the controller, thus the shield must be defined w.r.t. an abstraction of the plant, which we define formally below.

Abstraction. An abstraction is a *Markov decision process* (MDP, for short) is $\mathcal{A} = \langle \Gamma, A, a_0, \delta \rangle$, where Γ is a set of actions, A is a set of abstract plant states, $a_0 \in A$ is an initial state, and $\delta : A \times \Gamma \rightarrow [0, 1]^A$ is a probabilistic transition function, i.e., for every $a \in A$ and $\gamma \in \Gamma$, we have $\sum_{a' \in A} \delta(a, \gamma)(a') = 1$. The probabilities in the abstraction model our lack of knowledge of the plant, and we assume that they reflect the behavior exhibited by the plant. A *policy* f is a function from a finite history of states in A^* to the next action in Γ , thus it gives rise to a probabilistic distribution $\mathcal{D}(f)$ over infinite sequences over A .

Example 1. Consider a plant that represents a junction with four incoming directions (see Fig. 1). We describe an abstraction \mathcal{A} for the junction that specifies how many cars are waiting in each queue, where we cut off the count at a parameter $k \in \mathbb{N}$. Formally, an abstract state is a vector in $\{0, \dots, k\}^4$, where the indices respectively represent the North, East, South, and West queues. The larger k is, the closer the abstraction is to the concrete plant. The set of possible actions represent the possible light directions in the junction $\{\text{NS}, \text{EW}\}$. The abstract transitions estimate the plant behavior, and

we describe them in two steps. Consider an abstract state $a = (a_1, a_2, a_3, a_4)$ and suppose the issued action is NS, where the case of EW is similar. We allow a car to cross the junction from each of the North and South queues and decrease the two queues. Let $a' = (\max\{0, a_1 - 1\}, a_2, \max\{0, a_3 - 1\}, a_4)$. Next, we probabilistically model incoming cars to the queues as follows. Consider a vector $\langle i_1, i_2, i_3, i_4 \rangle \in \{0, 1\}^4$ that represents incoming cars to the queues. Let a'' be such that, for $1 \leq j \leq 4$, we add i_j to the j -th queue and trim at k , thus $a''_j = \min\{a'_j + i_j, k\}$. Then, in \mathcal{A} , when performing action NS in a , we move to a'' with the uniform probability $1/16$. \square

We define shields formally. Let Γ be a set of commands, M a set of memory states, C and A be a set of concrete and abstract states, respectively, and let $\alpha : C \rightarrow A$ be a mapping between the two. A shield is a function $\text{SHIELD} : A \times M \times \Gamma \rightarrow \Gamma \times M$ together with an initial memory state $m_0 \in M$. We use PLANT to refer to the plant, which, recall, has two functionalities: reading the current state and issuing a command from Γ . Let CONT be a controller, which has a single functionality: given a history of plant states, the controller issues the command to issue to the plant. The interaction of the components is captured in the following pseudo code:

```

 $m \leftarrow m_0 \in M$  and  $\pi \leftarrow$  empty sequence.
while true do
     $c \leftarrow \text{PLANT.GETSTATE}() \in C$ 
     $\gamma \leftarrow \text{CONT.GETCOMMAND}(\pi \cdot c)$ 
     $a = \alpha(c) \in A$  // generate abstract state for shield
     $\gamma', m' \leftarrow \text{SHIELD}(a, \gamma, m)$ 
     $\text{PLANT.ISSUECOMMAND}(\gamma')$ 
     $m \leftarrow m'$  // update shield memory state
     $\pi \leftarrow \pi \cdot \langle c, \gamma' \rangle$  // update plant history
end while

```

2.2 Quantitative Objectives for Shields

We are interested in two types of performance measures for shields. The *behavioral measure* quantifies the quality of the plant's behavior when operated with a controller and shield. The *interference measure* quantifies the degree to which a shield interferes with the controller. Formally, we need to specify values for infinite sequences, and we use *weighted automata*, which are a convenient model to express such values.

Weighted Automata. A weighted automaton is a function from infinite strings to values. Technically, a weighted automaton is similar to a standard automaton only that the transitions are labeled, in addition to letters, with numbers (weights). Unlike standard automata in which a run is either accepting or rejecting, a run in a weighted automaton has a value. We focus on limit-average automata in which the value is the limit average of the running sum of weights that it traverses. Formally, a weighted automaton is $\mathcal{W} = \langle \Sigma, Q, q_0, \Delta, \text{cost} \rangle$, where Σ is a finite alphabet, Q is a finite set of states, $\Delta \subseteq (Q \times \Sigma \times Q)$ is a deterministic transition relation, i.e., for every $q \in Q$ and $\sigma \in \Sigma$, there is at most one $q' \in Q$ with $\Delta(q, \sigma, q')$, and $\text{cost} : \Delta \rightarrow \mathbb{Q}$ specifies costs for transitions. A *run* of \mathcal{W} on an infinite word $\sigma = \sigma_1, \sigma_2, \dots \in Q^\omega$

such that $r_0 = q_0$ and, for $i \geq 1$, we have $\Delta(r_{i-1}, \sigma_i, r_i)$. Note that \mathcal{W} is deterministic so there is at most one run on every word. The value that \mathcal{W} assigns to σ is $\liminf_{n \rightarrow \infty} \frac{1}{n} \sum_{i=1}^n \text{cost}(r_{i-1}, \sigma_i, r_i)$.

Behavioral Score. A *behavioral* score measures the quality of the behavior that the plant exhibits. It is given by a weighed automaton over the alphabet $A \times \Gamma$, thus it assigns real values to infinite sequences over $A \times \Gamma$. In our experiments, we use a *concrete* behavioral score, which assigns values to infinite sequences over $C \times \Gamma$. We compare the performance of the plant with various controllers and shields w.r.t. the concrete score rather than the abstract score. With a weighted automaton we can express costs that change over time: for example, we can penalize traffic lights that change frequently.

Interference Score. The second score we consider measures the interference of the shield with the controller. An *interference* score is given by a weighted automaton over the alphabet $\Gamma \times \Gamma$. With a weighted automaton we can express costs that change over time: for example, interfering once costs 1 and any successive interference costs 2, thus we reward the shield for short interferences.

From Shields and Controllers to Policies. Consider an abstraction MDP \mathcal{A} . To ensure worst-case guarantees, we treat the controller as an adversary for the shield. Let SHIELD be a shield with memory set M and initial memory state m_0 . Intuitively, we find a policy in \mathcal{A} that represents the interaction of SHIELD with a controller that maximizes the cost incurred. Formally, an *abstract controller* is a function $\chi : A^* \rightarrow \Gamma$. The interaction between SHIELD and χ gives rise to a policy $\text{pol}(\text{SHIELD}, \chi)$ in \mathcal{A} , which, recall, is a function from A^* to Γ . We define $\text{pol}(\text{SHIELD}, \chi)$ inductively as follows. Consider a history $\pi \in A^*$ that ends in $a \in A$, and suppose the current memory state of SHIELD is $m \in M$. Let $\gamma = \chi(\pi)$ and let $\langle \gamma', m' \rangle = \text{SHIELD}(\gamma, a, m)$. Then, the action that the policy $\text{pol}(\text{SHIELD}, \chi)$ assigns is γ' , and we update the memory state to be m' .

Problem Definition; Quantitative Shield Synthesis Consider an abstraction MDP \mathcal{A} , a behavioral score BEH, an interference score INT, both given as weighted automata, and a factor $\lambda \in [0, 1]$ with which we weigh the two scores. Our goal is to find an *optimal shield* w.r.t. these inputs as we define below. Consider a shield SHIELD with memory set M . Let X be the set of abstract controllers. For SHIELD and $\chi \in X$, let $\mathcal{D}(\text{SHIELD}, \chi)$ be the probability distribution over $A \times \Gamma \times \Gamma$ that the policy $\text{pol}(\text{SHIELD}, \chi)$ gives rise to. The *value* of SHIELD, denoted $\text{val}(\text{SHIELD})$, is $\sup_{\chi \in X} \mathbb{E}_{r \sim \mathcal{D}(\text{SHIELD}, \chi)} [\lambda \cdot \text{INT}(r) + (1 - \lambda) \cdot \text{BEH}(r)]$. An *optimal shield* is a shield whose value is $\inf_{\text{SHIELD}} \text{val}(\text{SHIELD})$.

Remark 1 (Robustness and flexibility). The problem definition we consider allows quantitative optimization of shields w.r.t. two dimensions of quantitative measures. Earlier works have considered shields but mainly with respect to Boolean measures in both dimensions. For example, in [30], shields for safety behavioral measures were constructed with a Boolean notion of interference, as well as a Boolean notion of shield correctness. In contrast we allow quantitative objectives in both dimensions which presents a much more general and robust framework. For example, the first measure of correctness can be quantitative and minimize the error rate, and the second measure can allow

shields to correct but minimize the long-run average interference. Both of the above allows the shield to be flexible. Moreover, tuning the parameter λ allows flexible trade-off between the two.

We allow a robust class of quantitative specifications using weighted automata, which have been already established as a robust specification framework. Any automata model can be used in the framework, not necessarily the ones we use here. For example, weighted automata that discount the future or process only finite-words are suitable for planning purposes [32]. Thus our framework is a very robust and flexible framework for quantitative shield synthesis. \square

2.3 Examples

In Remark 1 we already discussed the flexibility of the framework. We now present concrete examples of instantiations of the optimization problem above on our running example, which illustrate how quantitative shields can be used to cope with limitations of learned controllers.

Dealing with Unexpected Plant Behavior; Rush-Hour Traffic. Consider the abstraction described in Example 1, where each abstract state is a 4-dimensional vector that represents the number of waiting cars in each direction. The behavioral score we use is called the *max queue*. It charges an abstract state $a \in \{0, \dots, k\}^4$ with the size of the maximal queue, no matter what the issued action is, thus $\text{cost}_{\text{BEH}}(a) = \max_{i \in \{1, 2, 3, 4\}} a_i$. A shield that minimizes the max-queue cost will prioritize the direction with the largest queue. For the interference score, we use a score that we call the *basic* interference score; we charge the shield 1 whenever it changes the controller's action and otherwise we charge it 0, and take the long-run average of the costs. Recall that in the construction in Example 1, we chose uniformly at random the vector of incoming cars. Here, in order to model rush-hour traffic, we use a different distribution, where we let p_j be the probability that a car enters the j -th queue. Then, the probability of a vector $\langle i_1, i_2, i_3, i_4 \rangle \in \{0, 1\}^4$ is $\prod_{1 \leq j \leq 4} (p_j \cdot i_j + (1 - p_j) \cdot (1 - i_j))$. To model a higher load traveling on the North-South route, we increase p_1 and p_3 beyond 0.5.

Weighing Different Goals; Local Fairness. Suppose the controller is trained to maximize the number of cars passing a *city*. Thus, it aims to maximize the speed of the cars in the city and prioritizes highways over farm roads. A secondary objective for a controller is to minimize local queues. Rather than adding this objective in the training phase, which can have an un-expected outcome, we can add a local shield for each junction. To synthesize the shield, we use the same abstraction and basic interference score as in the above. The behavioral score we use charges an abstract state $a \in \{0, \dots, k\}^4$ with difference $|(a_1 + a_3) - (a_2 + a_4)|$, thus the greater the inequality between the two waiting directions, the higher the cost.

Adding Features to the Controller; Prioritizing Public Transport. Suppose a controller is trained to increase throughput in a junction. After the controller is trained, a designer wants to add a functionality to the controller that prioritizes buses over personal vehicles. That is, if a bus is waiting in the North direction, and no bus is waiting in either the East or West directions, then the light should be North-South, and the other

cases are similar. The abstraction we use is simpler than the ones above since we only differentiate between a case in which a bus is present or not, thus the abstract states are $\{0, 1\}^4$, where the indices represent the directions clockwise starting from North. Let $\gamma = \text{NS}$. The behavioral cost of a state a is 1 when $a_2 = a_4 = 0$ and $a_1 = 1$ or $a_3 = 1$. The interference score we use is the basic one. A shield guarantees that in the long run, the specification is essentially never violated.

3 A Game-Theoretic Approach to Quantitative Shield Synthesis

In order to synthesize optimal shields we construct a two-player stochastic game [10], where we associate Player 2 with the shield and Player 1 with the controller. The game is defined on top of an abstraction and the players' objectives are given by the two performance measures. We first formally define stochastic games, then we construct the shield synthesis game, and finally show how to extract a shield from a strategy for Player 2.

Stochastic Graph Games. The game is played on a graph by placing a token on a vertex and letting the players move it throughout the graph. For ease of presentation, we fix the order in which the players move: first, Player 1, then Player 2, and then “Nature”, i.e., the next vertex is chosen randomly. Edges have costs, which, again for convenience, appear only on edges following Player 2 moves. Formally, a two-player stochastic graph-game is $\langle V_1, V_2, V_N, E, \Pr, \text{cost} \rangle$, where $V = V_1 \cup V_2 \cup V_N$ is a finite set of vertices that is partitioned into three sets, for $i \in \{1, 2\}$, Player i controls the vertices in V_i and “Nature” controls the vertices in V_N , $E \subseteq (V_1 \times V_2) \cup (V_2 \times V_N)$ is a set of deterministic edges, $\Pr : V_N \times V_1 \rightarrow [0, 1]$ is a probabilistic transition function, and $\text{cost} : (V_2 \times V_N) \rightarrow \mathbb{Q}$. Suppose the token reaches $v \in V$. If $v \in V_i$, for $i \in \{1, 2\}$, then Player i chooses the next position of the token $u \in V$, such that $E(v, u)$. If $v \in V_N$, then the next position is chosen randomly; namely, the token moves to $u \in V$ with probability $\Pr[v, u]$.

The game is a *zero-sum game*; Player 1 tries to maximize the expected long-run average of the accumulated costs, and Player 2 tries to minimize it. A *strategy* for Player i , for $i \in \{1, 2\}$, is a function that takes a history in $V^* \cdot V_i$ and returns the next vertex to move the token to. The games we consider admit *memoryless* optimal strategies, thus it suffices to define a Player i strategy as a function from V_i to V . We associate a *payoff* with two strategies f_1 and f_2 , which we define next. Given f_1 and f_2 , it is not hard to construct a Markov chain \mathcal{M} with states V_N and with weights on the edges: for $v, u \in V_N$, the probability of moving from v to u in \mathcal{M} is $\Pr_{\mathcal{M}}[v, u] = \sum_{w \in V_1 : f_2(f_1(w))=u} \Pr[v, w]$ and the cost of the edge is $\text{cost}_{\mathcal{M}}(v, u) = \sum_{w \in V_1 : f_2(f_1(w))=u} \Pr[v, w] \cdot \text{cost}(f_1(w), u)$. The *stationary distribution* s_v of a vertex $v \in V_N$ in \mathcal{M} is a well known concept [43] and it intuitively measures the long-run average time that is spent in v . The payoff w.r.t. f_1 and f_2 , denoted $\text{payoff}(f_1, f_2)$ is $\sum_{v, u \in V_N} s_v \cdot \Pr_{\mathcal{M}}[v, u] \cdot \text{cost}_{\mathcal{M}}(v, u)$. The payoff of a strategy is the payoff it guarantees against any strategy of the other player, thus $\text{payoff}(f_1) = \inf_{f_2} \text{payoff}(f_1, f_2)$. A strategy is *optimal* for Player 1 if it achieves the optimal payoff, thus f is optimal if $\text{payoff}(f) = \sup_{f_1} \text{payoff}(f_1)$. The definitions for Player 2 are dual.

Constructing the Synthesis Game. Consider an abstraction MDP $\mathcal{A} = \langle \Gamma, A, a_0, \delta \rangle$, weighted automata for the behavioral score $\text{BEH} = \langle A \times \Gamma, Q_{\text{BEH}}, q_0^{\text{BEH}}, \Delta_{\text{BEH}}, \text{cost}_{\text{BEH}} \rangle$ and interference score $\text{INT} = \langle \Gamma \times \Gamma, Q_{\text{INT}}, q_0^{\text{INT}}, \Delta_{\text{INT}}, \text{cost}_{\text{INT}} \rangle$, and a factor $\lambda \in [0, 1]$. We associate Player 1 with the controller and Player 2 with the shield. In each step, the controller first chooses an action, then the shield chooses whether to alter it, and the next state is selected at random. Let $S = A \times Q_{\text{INT}} \times Q_{\text{BEH}}$. We define $\mathcal{G}_{\mathcal{A}, \text{BEH}, \text{INT}, \lambda} = \langle V_1, V_2, V_N, E, \Pr, \text{cost} \rangle$, where

- $V_1 = S$,
- $V_2 = S \times \Gamma$,
- $V_N = S \times \Gamma \times \{N\}$, where the purpose of N is to differentiate between the vertices,
- $E(s, \langle s, \gamma \rangle)$
 - for $s \in S$ and $\gamma \in \Gamma$, and $E(\langle s, \gamma \rangle, \langle s', \gamma', N \rangle)$ for $s = \langle a, q_1, q_2 \rangle \in S$, $\gamma, \gamma' \in \Gamma$, and $s' = \langle a, q'_1, q'_2 \rangle \in S$ s.t. $\Delta_{\text{INT}}(q_1, \langle \gamma, \gamma' \rangle, q'_1)$ and $\Delta_{\text{BEH}}(q_2, \langle a, \gamma' \rangle, q'_2)$,
- $\Pr[\langle \langle a, q_1, q_2 \rangle, \gamma, N \rangle, \langle a', q_1, q_2 \rangle] = \delta(a, \gamma)(a')$, and
- for $s = \langle a, q_1, q_2 \rangle$ and $s' = \langle a, q'_1, q'_2 \rangle$ as in the above, we have $\text{cost}(\langle s, \gamma \rangle, \langle s', \gamma', N \rangle) = \lambda \cdot \text{cost}_{\text{INT}}(q_1, \langle \gamma, \gamma' \rangle, q'_1) + (1 - \lambda) \cdot \text{cost}_{\text{BEH}}(q_2, \langle \gamma', a \rangle, q'_2)$.

From Strategies to Shields. Recall that the game $\mathcal{G}_{\mathcal{A}, \text{BEH}, \text{INT}, \lambda}$ admits memoryless optimal strategies. Consider an optimal memoryless strategy f for Player 2. Thus, given a Player 2 vertex in V_2 , the function f returns a vertex in V_N to move to. The shield SHIELD_f that is associated with f has the memory set $M = Q_{\text{INT}} \times Q_{\text{BEH}}$ and the initial memory state is $\langle q_0^{\text{INT}}, q_0^{\text{BEH}} \rangle$. Given an abstract state $a \in A$, a memory state $\langle q_{\text{INT}}, q_{\text{BEH}} \rangle \in M$, and a controller action $\gamma \in \Gamma$, let $\langle a, q'_{\text{INT}}, q'_{\text{BEH}}, \gamma' \rangle = f(a, q_{\text{INT}}, q_{\text{BEH}}, \gamma)$. The shield SHIELD_f returns the action γ' and the updated memory state $\langle q'_{\text{INT}}, q'_{\text{BEH}} \rangle$.

Theorem 1. *Given an abstraction \mathcal{A} , weighted automata BEH and INT, and a factor λ , the game $\mathcal{G}_{\mathcal{A}, \text{BEH}, \text{INT}, \lambda}$ admits optimal memoryless strategies. Let f be an optimal memoryless strategy for Player 2. The shield SHIELD_f is an optimal shield w.r.t. \mathcal{A} , BEH, INT, and λ .*

Remark 2 (Shield size). Recall that a shield is a function $\text{SHIELD} : A \times \Gamma \times M \rightarrow \Gamma \times M$, which we store as a table. The *size* of the shield is the size of the domain, namely the number of entries in the table. Given an abstraction with n_1 states, a set of possible commands Γ , and weighted automata with n_2 and n_3 states, the size of the shield we construct is $n_1 \cdot n_2 \cdot n_3 \cdot |\Gamma|$. \square

Remark 3. Our construction of the game can be seen as a two-step procedure: we construct a stochastic game with two mean-payoff objectives, a.k.a. a *two-dimensional* game, where the shield player's goal is to minimize both the behavioral and interference scores separately. We then reduce the game to a “one-dimension” game by weighing the scores with the parameter λ . We perform this reduction for several reasons. First, while multi-dimensional quantitative objectives have been studied in several cases, such as MDPs [4,6,7] and special problems of stochastic games (e.g., almost-sure winning) [2,5,8], there is no general algorithmic solution known for stochastic games with two-dimensional objectives. Second, even for non-stochastic games with

two-dimensional quantitative objectives, infinite-memory is required in general [48]. Finally, in our setting, the parameter λ provides a meaningful tradeoff: it can be associated with how well we value the quality of the controller. If the controller is of poor quality, then we charge the shield less for interference and set λ to be low. On the other hand, for a high-quality controller, we charge the shield more for interferences and set a high value for λ . \square

4 Case Study

We experiment with our framework in designing quantitative shields for traffic-light controllers that are trained using reinforcement-learning (RL). We illustrate the usefulness of shields in dealing with limitations of RL as well as providing an intuitive framework to complement RL techniques.

Traffic Simulation. All experiments were conducted using traffic simulator “Simulation of Urban MObility” (SUMO, for short) [31] v0.22 using the SUMO Python API. Incoming traffic in the cities is chosen randomly. The simulations were executed on a desktop computer with a 4 x 2.70 GHz Intel Core i7-7500U CPU, 7.7 GB of RAM running Ubuntu 16.04.

The Traffic Light Controller. We use RL to train a city-wide traffic-signal controller. Intuitively, the controller is aware of the waiting cars in each junction and its actions constitute a light assignment to all the junctions. We train a controller using a deep convolutional Q-network [37]. In most of the networks we test with, there are two controlled junctions. The input vector to the neural network is a 16-dimensional vector, where 8 dimensions represent a junction. For each junction, the first four components state the number of cars approaching the junction and the last four components state the accumulated waiting time of the cars in each one of the lanes. For example, in Fig. 1, the first four components are (3, 6, 3, 1), thus the controller’s state is not trimmed at 5. The controller is trained to minimize both the number of cars waiting in the queues and the total waiting time. For each junction i , the controller can choose to set the light to be either NS_i or EW_i , thus the set of possible actions is $\Gamma = \{NS_1NS_2, EW_1NS_2, NS_1EW_2, EW_1EW_2\}$.

We use a network consisting of 4 layers: The input layer is a convolutional layer with 16 nodes, the first hidden and the second hidden layers consisting out of 604 nodes and 1166 nodes, respectively. The output layer consists of 4 neurons with linear activation functions, each representing one of the above mentioned actions listed in Γ . The Q-learning uses the learning rate $\alpha = 0.001$ and the discount factor 0.95 for the Q-update and an ϵ -greedy exploration policy. The artificial neural network is built on an open source implementation¹ using Keras [9] and additional optimized functionality was provided by the NumPy [40] library. We train for 100 training epochs, where each epoch is 1500 seconds of simulated traffic, plus 2000 additional seconds in which no new cars are introduced. The total training time of the agent is roughly 1.5 hours. While the RL procedure that we use is simple procedure, it is inspired by standard approaches

¹ <https://github.com/Wert1996/Traffic-Optimisation>.

to learning traffic controllers and produces controllers that perform relatively well also with no shield.

The Shield. We synthesize a “local” shield for a junction and copy the shield for each junction in the city. Recall that the first step in constructing the synthesis game is to construct an abstraction of the plant, which intuitively represents the information according to which the shield makes its decisions. The abstraction we use is described in Example 1; each state is a 4-dimensional integer in $\{0, \dots, k\}$, which represents an abstraction of the number of waiting cars in each direction, cut-off by $k \in \mathbb{N}$. As elaborated in the example, when a shield assigns a green light to a direction, we evict a car from the two respectable queues, and select the incoming cars uniformly at random. Regarding objectives, in most of our experiments, the behavioral score we use charges an abstract state $a \in \{0, \dots, k\}^4$ with $|a_1 + a_3 - (a_2 + a_4)|$, thus the shield aims to balance the total number of waiting cars per direction. The interference score we use charges the shield 1 for altering the controller’s action.

Since we use simple automata for objectives, the size of the shields we use is $|A \times \Gamma|$, where $|\Gamma| = 2$. In our experiments, we cut-off the queues at $k = 6$, which results in a shield of size 2592. The synthesis procedure’s running time is in the order of minutes. We have already pointed out that we are interested in small light-weight shields, and this is indeed what we construct. In terms of absolute size, the shield takes ~ 60 KB versus the controller who takes ~ 3 MB; a difference of 2 orders of magnitude.

Our synthesis procedure includes a solution to a stochastic mean-payoff game. The complexity of solving such games is an interesting combinatorial problem in NP and coNP (thus unlikely to be NP-hard) for which the existence of a polynomial-time algorithm is major long-standing open problem. The current best-known algorithms are exponential, and even for special cases like turn-based deterministic mean-payoff games or turn-based stochastic games with reachability objectives, no polynomial-time algorithms are known. The algorithm we implemented is called the *strategy iteration* algorithm [22, 23] in which one starts with a strategy and iteratively improves it, where each iteration requires polynomial time. While the algorithm’s worst-case complexity is exponential, in practice, the algorithm has been widely observed to terminate in a few number of iterations.

Evaluating Performance. Throughout all our experiments, we use a unified and concrete measure of performance: the total waiting time of the cars in the city. Our assumption is that minimizing this measure is the main objective of the designer of the traffic light system for the city. While performance is part of the objective function when training the controller, the other components of the objective are used in order to improve training. Similarly, the behavioral measure we use when synthesizing shields is chosen heuristically in order to construct shields that improve concrete performance.

The Effect of Changing λ . Recall that we use $\lambda \in [0, 1]$ in order to weigh between the behavioral and interference measures of a shield, where the larger λ is, the more the shield is charged for interference. In our first set of experiments, we fix all parameters apart from λ and synthesize shields for a city that has two controllable junctions. In the first experiment, we use a random traffic flow that is similar to the one used in training.

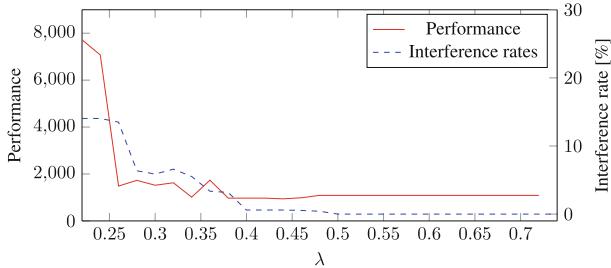


Fig. 2. Results for shields constructed with various λ values, together with a fixed plant and controller, where the simulation traffic distribution matches the one the controller is trained for.

We depict the results of the simulation in Fig. 2. We make several observations on the results below.

Interference. We observe that the ratio of the time that the shield intervenes is low: for most values of λ the ratio is well below 10%. For large values of λ , interference is too costly, and the shields become *trivial*, namely it never alters the actions of the controller. The performance we observe is thus the performance of the controller with no shield. In this set of experiments, we observe that the threshold after which shields become trivial is $\lambda = 0.5$, and for different setups, the threshold changes.

Performance. We observe that performance as function of λ , is a curve-like function. When λ is small, altering commands is cheap, the shield intervenes more frequently, and performance drops. This performance drop is expected: the shield is a simple device and the quality of its routing decisions cannot compete with the trained controller. This drop is also encouraging since it illustrates that our experimental setting is interesting. Surprisingly, we observe that the curve is in fact a paraboloid: for some values, e.g., $\lambda = 0.4$, the shield improves the performance of the controller. We find it unexpected that the shield improves performance even when observing trained behavior, and this performance increase is observed more significantly in the next experiments.

Rush-Hour Traffic. In Fig. 3, we use a shield to add robustness to a controller for behavior it was not trained for. We see a more significant performance gain in this exper-

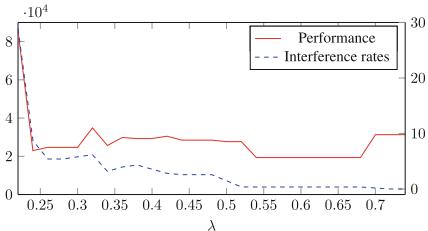


Fig. 3. Similar to Fig. 2 only that the simulation traffic distribution models rush-hour traffic.

Fig. 4. Comparing the variability in performance of the different controllers, with shield (blue) and without a shield (red). (Color figure online)



iment. We use the controller from the previous experiment, which is trained for uniform car arrival. We simulate it in a network with “rush-hour” traffic, which we model by significantly increasing the traffic load in the North-South direction. We synthesize shields that prefer to evict traffic from the North-South queue over the East-West queue. We achieve this by altering the objective in the stochastic game; we charge the shield a greater penalty for cars waiting in these queues over the other queues. For most values of λ below 0.7, we see a performance gain. Note that the performance of the controller with no shield is depicted on the far right, where the shield is trivial. An alternative approach to synthesize a shield would be to alter the probabilities in the abstraction, but we found that altering the weights results in a better performance gain.

Reducing Variability. Machine learning techniques are intricate, require expertise, and a fine tuning of parameters. This set of experiments show how the use of shields reduces variability of the controllers, and as a result, it reduces the importance of choosing the optimal parameters in the training phase. We fix one of the shields from the first experiment with $\lambda = 0.4$. We observe performance in a city with various controllers, which are trained with varying training parameters, when the controllers are run with and without the shield and on various traffic conditions that sometimes differ from the ones they are trained on.

The city we experiment with consists of a main two-lane road that crosses the city from East to West. The main road has two junctions in which smaller “farm roads” meet the main road. We refer to the *bulk traffic* as the traffic that only “crosses the city”; namely, it flows only on the main road either from East to West or in the opposite direction. For $r \in [0, 1]$, Controller- r is trained where the ratio of the bulk traffic out of the total traffic is r . That is, the higher r is, the less traffic travels on the farm roads. We run simulations in which Controller- r observes bulk traffic $k \in [0, 1]$, which it was not necessarily trained for.

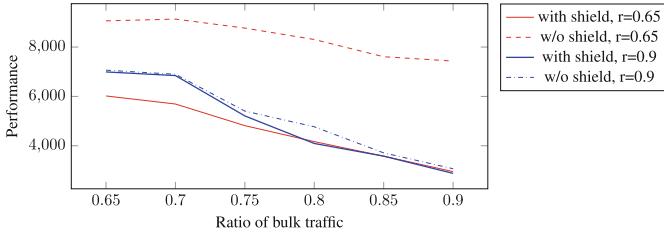


Fig. 5. Results for Controllers-0.65 and 0.9 exhibiting traffic that they are not trained for, with and without a shield. Performance is the total waiting time of the cars in the city.

In Fig. 5, we depict the performance of two controllers for various traffic settings. We observe, in these two controllers as well as the others, that operating with a shield consistently improves performance. The plots illustrate the unexpected behavior of machine-learning techniques: e.g., when run without a shield, Controller-0.9 outperforms Controller-0.65 in all settings, even in the setting 0.65 on which Controller-0.65 was trained on. Thus, a designer who expects a traffic flow of 0.65, would be better

off training with a traffic of 0.9. A shield improves performance and thus reduces the importance of which training data to use.

Measuring Variability. In Fig. 4, we depict the variability in performance between the controllers. The higher the variability is, the more significant it is to choose the right parameters when training the controller. Formally, let $R = \{0.65, 0.7, 0.75, 0.8, 0.85, 0.9\}$. For $r, k \in R$, we let $\text{Perf}(r, k)$ denote the performance (total waiting times) when Controller- r observes bulk traffic k . For each $k \in R$, we depict $\max_{r \in R} \text{Perf}(r, k) - \min_{r' \in R} \text{Perf}(r', k)$, when operating with and without a shield.

Clearly, the variability with a shield is significantly lower than without one. This data shows that when operating with a shield, it does not make much difference if a designer trains a controller with setting r or r' . When operating without a shield, the difference is significant.

Overcoming Liveness Bugs. Finding bugs in learned controllers is a challenging task. Shields bypass the need to find bugs since they treat the controller as a black-box and correct its behavior. We illustrate their usefulness in dealing with liveness bugs. In the same network as in the previous setting, we experiment with a controller whose training process lacked variability. In Fig. 6, we depict the light configuration throughout the experiment on the main road; the horizontal axis represents time, red means a red light for the main road and dually green. Initially, the controller performs well, but roughly half-way through the simulation it hits a bad state after which the light stays red. The shield, with only a few interferences, which are represented with dots, manages to recover the controller from its stuck state. In Fig. 7, we depict the number of waiting cars in the city, which clearly skyrockets once the controller gets stuck. It is evident that initially, the controller performs well. This point highlights that it is difficult to recog-



Fig. 6. The light in the East-West direction (the main road) of a junction. On bottom, with no shield the controller is stuck. On top, the shield's interferences are marked with dots.

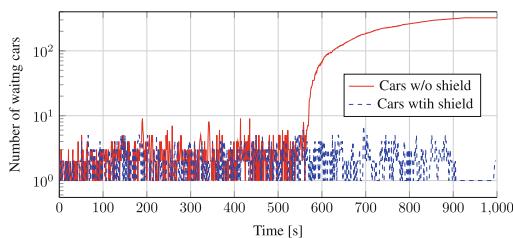


Fig. 7. The total number of waiting cars (log-scale) with and without a shield. Initially, the controller performs well on its own, until it gets stuck and traffic in the city freezes.

nize when a controller has a bug – in order to catch such a bug, a designer would need to find the right simulation and run it half way through before the bug appears.

One way to regain liveness would be to synthesize a shield for the qualitative property “each direction eventually gets a green light”. Instead, we use a shield that is synthesized for the quantitative specification as in the previous experiment. The shield, with a total of only 20 alterations is able to recover the controller from the bad state it is stuck in, and traffic flows correctly.

Adding Functionality; Prioritizing Public Transport. Learned controllers are monolithic. Adding functionality to a controller requires a complete re-training, which is time consuming, computationally costly, and requires care; changes in the objective can cause unexpected side effects to the performance. We illustrate how, using a shield, we can add to an existing controller, the functionality of prioritizing public transport.

The abstraction over which the shield is constructed slightly differs from the one used in the other experiments. The abstract state space is the same, namely four-dimensional vectors, though we interpret the entries as the positions of a bus in the respective queue. For example, the state $(0, 3, 0, 1)$ represents no bus in the North queue and a bus which is waiting, third in line, in the East queue. Outgoing edges from an abstract state also differ as they take into account, using probability, that vehicles might enter the queues between buses. For the behavioral score, we charge an abstract state with the sum of its entries, thus the shield is charged whenever buses are waiting and it aims to evict them from the queues as soon as possible.

In Fig. 8, we depict the performance of all vehicles and only buses as a function of the weighing factor λ . The result of this experiment is positive; the predicted behavior is observed. Indeed, when λ is small, interferences are cheap, which increase bus performance at the expense of the general performance. The experiment illustrates that the parameter λ is a convenient method to control the degree of prioritization of buses.

Local Fairness. In this experiment, we add local fairness to a controller that was trained for a global objective. We experiment with a network with four junctions and a city-wide controller, which aims to minimize total waiting times. Figure 9 shows that when the controller is deployed on its own, queues form in the city whereas a shield, which was synthesized as in the first experiments, prevents such local queues from forming.

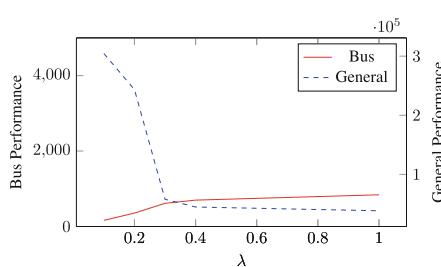


Fig. 8. The waiting time of buses/all vehicles with shields parameterized by λ .

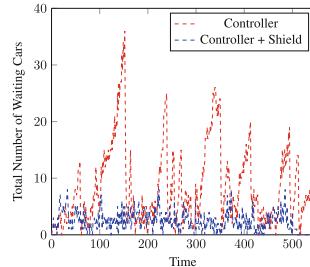


Fig. 9. Comparing the amount of waiting cars with and without a shield.

5 Discussion and Future Work

We suggest a framework for automatically synthesizing quantitative runtime shields to cope with limitations of machine-learning techniques. We show how shields can increase robustness to untrained behavior, deal with liveness bugs without verification, add features without retraining, and decrease variability of performance due to changes in the training parameters, which is especially helpful for machine learning non-experts. We use weighted automata to evaluate controller and shield behavior and construct a game whose solution is an optimal shield w.r.t. a weighted specification and a plant abstraction. The framework is robust and can be applied in any setting where learned or other black-box controllers are used.

We list several directions for further research. In this work, we make no assumptions on the controller and treat it adversarially. Since the controller might have bugs, modelling it as adversarial is reasonable. Though, it is also a crude abstraction since typically, the objectives of the controller and shield are similar. For future work, we plan to study ways to model the spectrum between cooperative and adversarial controllers together with solution concepts for the games that they give rise to.

In this work we make no assumptions on the relationship between the plant and the abstraction. While the constructed shields are optimal w.r.t. the given abstraction, the scores they guarantee w.r.t. the abstraction do not imply performance guarantees on the plant. To be able to produce performance guarantees on the concrete plant, we need guarantees on the relationship between the plant its abstraction. For future work, we plan to study the addition of such guarantees and how they affect the quality measures.

References

1. Alshiekh, M., Bloem, R., Ehlers, R., Könighofer, B., Niekum, S., Topcu, U.: Safe reinforcement learning via shielding. In: AAAI. AAAI Press (2018)
2. Basset, N., Kwiatkowska, M.Z., Wiltsche, C.: Compositional strategy synthesis for stochastic games with multiple objectives. *Inf. Comput.* **261**(Part), 536–587 (2018)
3. Bloem, R., Chatterjee, K., Jobstmann, B.: Graph games and reactive synthesis. In: Clarke, E., Henzinger, T., Veith, H., Bloem, R. (eds.) *Handbook of Model Checking*, pp. 921–962. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_27
4. Brázdil, T., Brozek, V., Chatterjee, K., Forejt, V., Kučera, A.: Two views on multiple mean-payoff objectives in Markov decision processes. *Log. Methods Comput. Sci.* **10**(1) (2014). <https://lmcs.episciences.org/1156>
5. Chatterjee, K., Doyen, L.: Perfect-information stochastic games with generalized mean-payoff objectives. In: Proceedings of the 31st LICS, pp. 247–256 (2016)
6. Chatterjee, K., Kretínská, Z., Kretínský, J.: Unifying two views on multiple mean-payoff objectives in Markov decision processes. *Log. Methods Comput. Sci.* **13**(2) (2017). <https://lmcs.episciences.org/3757>
7. Chatterjee, K., Majumdar, R., Henzinger, T. A.: Markov decision processes with multiple objectives. In: Proceedings of the 23rd STACS, pp. 325–336 (2006)
8. Chen, T., Forejt, V., Kwiatkowska, M. Z., Simaitis, A., Trivedi, A., Ummels, M.: Playing stochastic games precisely. In: Proceedings of the 23rd CONCUR, pp. 348–363 (2012)
9. Chollet, F.: keras (2015). <https://github.com/fchollet/keras>

10. Condon, A.: On algorithms for simple stochastic games. In: Proceedings of the DIMACS, pp. 51–72 (1990)
11. Dalal, G., Dvijotham, K., Vecerik, M., Hester, T., Paduraru, C., Tassa, Y.: Safe exploration in continuous action spaces. coRR, abs/1801.08757 (2017). [arXiv:1801.08757](https://arxiv.org/abs/1801.08757)
12. Desai, A., Ghosh, S., Seshia, S. A., Shankar, N., Tiwari, A.: SOTER: programming safe robotics system using runtime assurance. coRR, abs/1808.07921 (2018). [arXiv:1808.07921](https://arxiv.org/abs/1808.07921)
13. Dewey, D.: Reinforcement learning and the reward engineering principle. In: 2014 AAAI Spring Symposium Series (2014)
14. Droste, M., Kuich, W., Vogler, H.: Handbook of Weighted Automata. Springer, Heidelberg (2009). <https://doi.org/10.1007/978-3-642-01492-5>
15. Duan, Y., Chen, X., Houthooft, R., Schulman, J., Abbeel, P.: Benchmarking deep reinforcement learning for continuous control. In: Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, 19–24 June 2016, pp. 1329–1338 (2016)
16. Falcone, Y., Mounier, L., Fernandez, J.-C., Richier, J.-L.: Runtime enforcement monitors: composition, synthesis, and enforcement abilities. *Formal Methods Syst. Des.* **38**(3), 223–262 (2011)
17. Fulton, N., Platzer, A.: Safe reinforcement learning via formal methods: toward safe control through proof and learning. In: AAAI. AAAI Press (2018)
18. García, J., Fernández, F.: A comprehensive survey on safe reinforcement learning. *J. Mach. Learn. Res.* **16**, 1437–1480 (2015)
19. Geibel, P.: Reinforcement learning for MDPs with constraints. In: Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) ECML 2006. LNCS (LNAI), vol. 4212, pp. 646–653. Springer, Heidelberg (2006). https://doi.org/10.1007/11871842_63
20. Genders, W., Razavi, S.: Asynchronous n-step q-learning adaptive traffic signal control. *J. Intell. Trans. Syst.* **23**(4), 319–331 (2019)
21. Hamlen, K.W., Morrisett, J.G., Schneider, F.B.: Computability classes for enforcement mechanisms. *ACM Trans. Program. Lang. Syst.* **28**(1), 175–205 (2006)
22. Hoffman, A.J., Karp, R.M.: On nonterminating stochastic games. *Manag. Sci.* **12**(5), 359–370 (1966)
23. Howard, A.R.: Dynamic Programming and Markov Processes. MIT Press, Cambridge (1960)
24. Huang, X., Kwiatkowska, M., Wang, S., Wu, M.: Safety verification of deep neural networks. In: Proceedings of the 29th CAV, pp. 3–29 (2017)
25. Jansen, N., Könighofer, B., Junges, S., Bloem, R.: Shielded decision-making in MDPs. CoRR, [arXiv:1807.06096](https://arxiv.org/abs/1807.06096) (2018)
26. Ji, Y., Lafontaine, S.: Enforcing opacity by publicly known edit functions. In: 56th IEEE Annual Conference on Decision and Control, CDC 2017, Melbourne, Australia, 12–15 December 2017, pp. 4866–4871 (2017)
27. Kaelbling, L.P., Littman, M.L., Cassandra, A.R.: Planning and acting in partially observable stochastic domains. *Artif. Intell.* **101**(1), 99–134 (1998)
28. Kaelbling, L.P., Littman, M.L., Moore, A.W.: Reinforcement learning: a survey. *JAIR* **4**, 237–285 (1996)
29. Katz, G., Barrett, C.W., Dill, C. W., Julian, K., Kochenderfer, M.J.: Reluplex: an efficient SMT solver for verifying deep neural networks. In: Proceedings of the 29th CAV, pp. 97–117 (2017)
30. Könighofer, B., Alshiekh, M., Bloem, R., Humphrey, L., Könighofer, R., Topcu, U., Wang, C.: Shield synthesis. *Formal Methods Syst. Des.* **51**(2), 332–361 (2017)
31. Kratzewicz, D., Erdmann, J., Behrisch, M., Bieker, L.: Recent development and applications of SUMO - Simulation of Urban MOBility. *Int. J. Adv. Syst. Meas.* **5**(3&4), 128–138 (2012)

32. Lahijanian, M., Almagor, S., Fried, D., Kavraki, L.E., Vardi, M.Y.: This time the robot settles for a cost: a quantitative approach to temporal logic planning with partial satisfaction. In: Proceedings of the 29th AAAI, pp. 3664–3671 (2015)
33. Levine, S., Pastor, P., Krizhevsky, A., Ibarz, J., Quillen, D.: Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection. *I. J. Robot. Res.* **37**(4–5), 421–436 (2018)
34. Lillicrap, T.P.: Continuous control with deep reinforcement learning. arXiv preprint [arXiv:1509.02971](https://arxiv.org/abs/1509.02971) (2015)
35. Mannion, P., Duggan, J., Howley, E.: An experimental review of reinforcement learning algorithms for adaptive traffic signal control. In: McCluskey, T.L., Kotsialos, A., Müller, J.P., Klügl, F., Rana, O., Schumann, R. (eds.) Autonomic Road Transport Support Systems. AS, pp. 47–66. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-25808-9_4
36. Mason, G., Calinescu, R., Kudenko, D., Banks, A.: Assured reinforcement learning with formally verified abstract policies. In: Proceedings of the 9th International Conference on Agents and Artificial Intelligence, ICAART 2017, Porto, Portugal, 24–26 February 2017, vol. 2, pp. 105–117 (2017)
37. Mnih, V.: Human-level control through deep reinforcement learning. *Nature* **518**(7540), 529–533 (2015)
38. Moldovan, T.M., Abbeel, P.: Safe exploration in Markov decision processes. In: Proceedings of the 29th International Conference on Machine Learning, ICML 2012, Edinburgh, Scotland, UK, June 26 – July 1, 2012 (2012)
39. Mousavi, S.S., Schukat, M., Howley, E.: Traffic light control using deep policy-gradient and value-function-based reinforcement learning. *IET Intell. Trans. Syst.* **11**(7), 417–423 (2017)
40. Oliphant, T.E.: Guide to NumPy, 2nd edn. CreateSpace Independent Publishing Platform, USA (2015)
41. Phan, D., Yang, J., Grosu, R., Smolka, S.A., Stoller, S.D.: Collision avoidance for mobile robots with limited sensing and limited information about moving obstacles. *Formal Methods Syst. Des.* **51**(1), 62–86 (2017)
42. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, 11–13 January 1989, pp. 179–190 (1989)
43. Puterman, M.L.: *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons Inc., New York (2005)
44. Ramadge, P.J., Wonham, W.M.: Supervisory control of a class of discrete-event processes. *SIAM J. Control Optim.* **25**(1), 206–230 (1987)
45. Schneider, F.B.: Enforceable security policies. *ACM Trans. Inf. Syst. Secur.* **3**(1), 30–50 (2000)
46. Seshia, S. A., Sadigh, D.: Towards verified artificial intelligence. CoRR, [arXiv:1606.08514](https://arxiv.org/abs/1606.08514) (2016)
47. Sha, L.: Using simplicity to control complexity. *IEEE Soft.* **18**(4), 20–28 (2001)
48. Velner, Y., Chatterjee, K., Doyen, L., Henzinger, T.A., Rabinovich, A.M., Raskin, J.-F.: The complexity of multi-mean-payoff and multi-energy games. *Inf. Comput.* **241**, 177–196 (2015)
49. Wu, Y., Raman, V., Rawlings, B.C., Lafourche, S., Seshia, S.A.: Synthesis of obfuscation policies to ensure privacy and utility. *J. Autom. Reasoning* **60**(1), 107–131 (2018)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Taming Delays in Dynamical Systems

Unbounded Verification of Delay Differential Equations

Shenghua Feng^{1,2} , Mingshuai Chen^{1,2} , Naijun Zhan^{1,2} , Martin Fränzle³ , and Bai Xue^{1,2}

¹ SKLCS, Institute of Software, CAS, Beijing, China

{fengsh, chenms, znj, xuebai}@ios.ac.cn

² University of Chinese Academy of Sciences,
Beijing, China

³ Carl von Ossietzky Universität Oldenburg,
Oldenburg, Germany

fraenzle@informatik.uni-oldenburg.de



Abstract. Delayed coupling between state variables occurs regularly in technical dynamical systems, especially embedded control. As it consequently is omnipresent in safety-critical domains, there is an increasing interest in the safety verification of systems modelled by Delay Differential Equations (DDEs). In this paper, we leverage qualitative guarantees for the existence of an exponentially decreasing estimation on the solutions to DDEs as established in classical stability theory, and present a quantitative method for constructing such delay-dependent estimations, thereby facilitating a reduction of the verification problem over an unbounded temporal horizon to a bounded one. Our technique builds on the linearization technique of nonlinear dynamics and spectral analysis of the linearized counterparts. We show experimentally on a set of representative benchmarks from the literature that our technique indeed extends the scope of bounded verification techniques to unbounded verification tasks. Moreover, our technique is easy to implement and can be combined with any automatic tool dedicated to bounded verification of DDEs.

Keywords: Unbounded verification · Delay Differential Equations (DDEs) · Safety and stability · Linearization · Spectral analysis

1 Introduction

The theory of dynamical systems featuring delayed coupling between state variables dates back to the 1920s, when Volterra [41, 42], in his research on predator-prey models and viscoelasticity, formulated some rather general differential equations incorporating the past states of the system. This formulation, now known as delay differential equations (DDEs), was developed further by, e.g., Mishkis [30] and Bellman

This work has been supported through grants by NSFC under grant No. 61625206, 61732001 and 61872341, by Deutsche Forschungsgemeinschaft through grants No. GRK 1765 and FR 2715/4, and by the CAS Pioneer Hundred Talents Program under grant No. Y8YC235015.

and Cooke [2], and has witnessed numerous applications in many domains. Prominent examples include population dynamics [25], where birth rate follows changes in population size with a delay related to reproductive age; spreading of infectious diseases [5], where delay is induced by the incubation period; or networked control systems [21] with their associated transport delays when forwarding data through the communication network. These applications range further to models in optics [23], economics [38], and ecology [13], to name just a few. Albeit resulting in more accurate models, the presence of time delays in feedback dynamics often induces considerable extra complexity when one attempts to design or even verify such dynamical systems. This stems from the fact that the presence of feedback delays reduces controllability due to the impossibility of immediate reaction and enhances the likelihood of transient overshoot or even oscillation in the feedback system, thus violating safety or stability certificates obtained on idealized, delay-free models of systems prone to delayed coupling.

Though established automated methods addressing ordinary differential equations (ODEs) and their derived models, like hybrid automata, have been extensively studied in the verification literature, techniques pertaining to ODEs do not generalize straightforwardly to delayed dynamical systems described by DDEs. The reason is that the future evolution of a DDE is no longer governed by the current state instant only, but depends on a chunk of its historical trajectory, such that introducing even a single constant delay immediately renders a system with finite-dimensional states into an infinite-dimensional dynamical system. There are approximation methods, say the Padé approximation [39], that approximate DDEs with finite-dimensional models, which however may hide fundamental behaviors, e.g. (in-)stability, of the original delayed dynamics, as remarked in Sect. 5.2.2.8.1 of [26]. Consequently, despite well-developed numerical methods for solving DDEs as well as methods for stability analysis in the realm of control theory, hitherto in automatic verification, only a few approaches address the effects of delays due to the immediate impact of delays on the structure of the state spaces to be traversed by state-exploratory methods.

In this paper, we present a constructive approach dedicated to verifying safety properties of delayed dynamical systems encoded by DDEs, where the safety properties pertain to an infinite time domain. This problem is of particular interests when one pursues correctness guarantees concerning dynamics of safety-critical systems over a long run. Our approach builds on the *linearization* technique of potentially nonlinear dynamics and *spectral analysis* of the linearized counterparts. We leverage qualitative guarantees for the existence of an exponentially decreasing estimation on the solutions to DDEs as established in classical stability theory (see, e.g., [2, 19, 24]), and present a quantitative method to construct such estimations, thereby reducing the temporally unbounded verification problems to their bounded counterparts.

The class of systems we consider features delayed differential dynamics governed by DDEs of the form $\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), \mathbf{x}(t - r_1), \dots, \mathbf{x}(t - r_k))$ with initial states specified by a continuous function $\phi(t)$ on $[-r_{\max}, 0]$ where $r_{\max} = \max\{r_1, \dots, r_k\}$. It thus involves a combination of ODE and DDE with multiple constant delays $r_i > 0$, and has been successfully used to model various real-world systems in the aforementioned fields. In general, formal verification of unbounded safety or, dually, reachability properties of such systems inherits undecidability from similar properties for ODEs

(cf. e.g., [14]). We therefore tackle this unbounded verification problem by leveraging a stability criterion of the system under investigation.

Contributions. In this paper, we present a quantitative method for constructing a delay-dependent, exponentially decreasing upper bound, if existent, that encloses trajectories of a DDE originating from a certain set of initial functions. This method consequently yields a temporal bound T^* such that for any $T > T^*$, the system is safe over $[-r_{\max}, T]$ iff it is safe over $[-r_{\max}, \infty)$. For linear dynamics, such an equivalence of safety applies to any initial set of functions drawn from a compact subspace in \mathbb{R}^n ; while for nonlinear dynamics, our approach produces (a subset of) the *basin of attraction* around a *steady state*, and therefore a certificate (by bounded verification in finitely many steps) that guarantees the reachable set being contained in this basin suffices to claim safety/unsafety of the system over an infinite time horizon. Our technique is easy to implement and can be combined with any automatic tool for bounded verification of DDEs. We show experimentally on a set of representative benchmarks from the literature that our technique effectively extends the scope of bounded verification techniques to unbounded verification tasks.

Related Work. As surveyed in [14], the research community has over the past three decades vividly addressed automatic verification of hybrid discrete-continuous systems in a safety-critical context. The almost universal undecidability of the unbounded reachability problem, however, confines the sound key-press routines to either semi-decision procedures or even approximation schemes, most of which address bounded verification by computing the finite-time image of a set of initial states. It should be obvious that the functional rather than state-based nature of the initial condition of DDEs prevents a straightforward generalization of this approach.

Prompted by actual engineering problems, the interest in safety verification of continuous or hybrid systems featuring delayed coupling is increasing recently. We classify these contributions into two tracks. The first track pursues propagation-based bounded verification: Huang et al. presented in [21] a technique for simulation-based time-bounded invariant verification of nonlinear networked dynamical systems with delayed interconnections, by computing bounds on the sensitivity of trajectories to changes in initial states and inputs of the system. A method adopting the paradigm of verification-by-simulation (see, e.g., [9, 16, 31]) was proposed in [4], which integrates rigorous error analysis of the numeric solving and the sensitivity-related state bloating algorithms (cf. [7]) to obtain safe enclosures of time-bounded reachable sets for systems modelled by DDEs. In [46], the authors identified a class of DDEs featuring a local homeomorphism property which facilitates construction of over- and under-approximations of reachable sets by performing reachability analysis on the boundaries of the initial sets. Goubault et al. presented in [17] a scheme to compute inner- and outer-approximating flowpipes for DDEs with uncertain initial states and parameters using Taylor models combined with space abstraction in the shape of zonotopes. The other track of the literature tackles unbounded reachability problem of DDEs by taking into account the asymptotic behavior of the dynamics under investigation, captured by, e.g., Lyapunov functions in [32, 47] and barrier certificates in [35]. These approaches however share a common limitation that a polynomial template has to be specified either for the interval

Taylor models exploited in [47] (and its extension [29] to cater for properties specified as bounded metric interval temporal logic (MITL) formulae), for Lyapunov functionals in [32], or for barrier certificates in [35]. Our approach drops this limitation by resorting to the linearization technique followed by spectral analysis of the linearized counterparts, and furthermore extends over [47] by allowing immediate feedback (i.e. $\mathbf{x}(t)$) as well as multiple delays in the dynamics), to which their technique does not generalize immediately. In contrast to the *absolute stability* exploited in [32], namely a criterion that ensures stability for arbitrarily large delays, we give the construction of a delay-dependent stability certificate thereby substantially increasing the scope of dynamics amenable to stability criteria, for instance, the famous Wright's equation (cf. [44]). Finally, we refer the readers to [34] and [33] for related contributions in showing the existence of abstract symbolic models for nonlinear control systems with time-varying and unknown time-delay signals via approximate bisimulations.

2 Problem Formulation

Notations. Let \mathbb{N} , \mathbb{R} and \mathbb{C} be the set of natural, real and complex numbers, respectively. Vectors will be denoted by boldface letters. For $z = a + ib \in \mathbb{C}$ with $a, b \in \mathbb{R}$, the real and imaginary parts of z are denoted respectively by $\Re(z) = a$ and $\Im(z) = b$; $|z| = \sqrt{a^2 + b^2}$ is the modulus of z . For a vector $\mathbf{x} \in \mathbb{R}^n$, x_i refers to its i -th component, and its maximum norm is denoted by $\|\mathbf{x}\| = \max_{1 \leq i \leq n} |x_i|$. We define for $\delta > 0$, $\mathcal{B}(\mathbf{x}, \delta) = \{\mathbf{x}' \in \mathbb{R}^n \mid \|\mathbf{x}' - \mathbf{x}\| \leq \delta\}$ as the δ -closed ball around \mathbf{x} . The notation $\|\cdot\|$ extends to a set $X \subseteq \mathbb{R}^n$ as $\|X\| = \sup_{\mathbf{x} \in X} \|\mathbf{x}\|$, and to an $m \times n$ complex-valued matrix A as $\|A\| = \max_{1 \leq i \leq m} \sum_{j=1}^n |a_{ij}|$. \bar{X} is the closure of X and ∂X denotes the boundary of X . For $a \leq b$, let $C^0([a, b], \mathbb{R}^n)$ denote the space of continuous functions from $[a, b]$ to \mathbb{R}^n , which is associated with the maximum norm $\|f\| = \max_{t \in [a, b]} \|f(t)\|$. We abbreviate $C^0([-r, 0], \mathbb{R}^n)$ as \mathcal{C}_r for a fixed positive constant r , and let C^1 consist of all continuously differentiable functions. Given $f: [0, \infty) \mapsto \mathbb{R}$ a measurable function such that $\|f(t)\| \leq ae^{bt}$ for some constants a and b , then the Laplace transform $\mathcal{L}\{f\}$ defined by $\mathcal{L}\{f\}(z) = \int_0^\infty e^{-zt} f(t) dt$ exists and is an analytic function of z for $\Re(z) > b$.

Delayed Differential Dynamics. We consider a class of dynamical systems featuring delayed differential dynamics governed by DDEs of autonomous type:

$$\begin{cases} \dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), \mathbf{x}(t - r_1), \dots, \mathbf{x}(t - r_k)), & t \in [0, \infty) \\ \mathbf{x}(t) = \phi(t), & t \in [-r_k, 0] \end{cases} \quad (1)$$

where \mathbf{x} is the time-dependent *state* vector in \mathbb{R}^n , $\dot{\mathbf{x}}$ denotes its temporal derivative $d\mathbf{x}/dt$, and t is a real variable modelling time. The discrete delays are assumed to be ordered as $r_k > \dots > r_1 > 0$, and the initial states are specified by a vector-valued function $\phi \in \mathcal{C}_{r_k}$.

Suppose \mathbf{f} is a Lipschitz-continuous vector-valued function in $C^1(\mathbb{R}^{(k+1)n}, \mathbb{R}^n)$, which implies that the system has a unique maximal *solution* (or *trajectory*) from a given initial condition $\phi \in \mathcal{C}_{r_k}$, denoted as $\xi_\phi: [-r_k, \infty) \mapsto \mathbb{R}^n$. We denote in the

sequel by $\mathbf{f}_\mathbf{x} \hat{=} \left[\frac{\partial \mathbf{f}}{\partial x_1} \cdots \frac{\partial \mathbf{f}}{\partial x_n} \right]$ the Jacobian matrix (i.e., matrix consisting of all first-order partial derivatives) of \mathbf{f} w.r.t. the component $\mathbf{x}(t)$. Similar notations apply to components $\mathbf{x}(t - r_i)$, for $i = 1, \dots, k$.

Example 1 (Gene regulation [12, 36]). The control of gene expression in cells is often modelled with time delays in equations of the form

$$\begin{cases} \dot{x}_1(t) = g(x_n(t - r_n)) - \beta_1 x_1(t) \\ \dot{x}_j(t) = x_{j-1}(t - r_{j-1}) - \beta_j x_j(t), \quad 1 < j \leq n \end{cases} \quad (2)$$

where the gene is transcribed producing mRNA (x_1), which is translated into enzyme x_2 that in turn produces another enzyme x_3 and so on. The end product x_n acts to repress the transcription of the gene by $\dot{g} < 0$. Time delays are introduced to account for time involved in transcription, translation, and transport. The positive β_j 's represent decay rates of the species. The dynamic described in Eq. (2) falls exactly into the scope of systems considered in this paper, and in fact, it instantiates a more general family of systems known as monotone cyclic feedback systems (MCFS) [28], which includes neural networks, testosterone control, and many other effects in systems biology.

Lyapunov Stability. Given a system of DDEs in Eq. (1), suppose \mathbf{f} has a steady state (a.k.a., *equilibrium*) at \mathbf{x}_e such that $\mathbf{f}(\mathbf{x}_e, \dots, \mathbf{x}_e) = \mathbf{0}$ then

- \mathbf{x}_e is said to be *Lyapunov stable*, if for every $\epsilon > 0$, there exists $\delta > 0$ such that, if $\|\phi - \mathbf{x}_e\| < \delta$, then for every $t \geq 0$ we have $\|\xi_\phi(t) - \mathbf{x}_e\| < \epsilon$.
- \mathbf{x}_e is said to be *asymptotically stable*, if it is Lyapunov stable and there exists $\delta > 0$ such that, if $\|\phi - \mathbf{x}_e\| < \delta$, then $\lim_{t \rightarrow \infty} \|\xi_\phi(t) - \mathbf{x}_e\| = 0$.
- \mathbf{x}_e is said to be *exponentially stable*, if it is asymptotically stable and there exist $\alpha, \beta, \delta > 0$ such that, if $\|\phi - \mathbf{x}_e\| < \delta$, then $\|\xi_\phi(t) - \mathbf{x}_e\| \leq \alpha \|\phi - \mathbf{x}_e\| e^{-\beta t}$, for all $t \geq 0$. The constant β is called the *rate of convergence*.

Here \mathbf{x}_e can be generalized to a constant function in \mathcal{C}_{r_k} when employing the supremum norm $\|\phi - \mathbf{x}_e\|$ over functions. This norm further yields the *locality* of the above definitions, i.e., they describe the behavior of a system near an equilibrium, rather than of all initial conditions $\phi \in \mathcal{C}_{r_k}$, in which case it is termed the *global stability*. W.l.o.g., we assume $\mathbf{f}(\mathbf{0}, \dots, \mathbf{0}) = \mathbf{0}$ in the sequel and investigate the stability of the zero equilibrium thereof. Any nonzero equilibrium can be straightforwardly shifted to a zero one by coordinate transformation while preserving the stability properties, see e.g., [19].

Safety Verification Problem. Given $\mathcal{X} \subseteq \mathbb{R}^n$ a compact set of initial states and $\mathcal{U} \subseteq \mathbb{R}^n$ a set of unsafe or otherwise bad states, a delayed dynamical system of the form (1) is said to be *T-safe* iff all trajectories originating from any $\phi(t)$ satisfying $\phi(t) \in \mathcal{X}, \forall t \in [-r_k, 0]$ do not intersect with \mathcal{U} at any $t \in [-r_k, T]$, and *T-unsafe* otherwise. In particular, we distinguish *unbounded verification* with $T = \infty$ from *bounded verification* with $T < \infty$.

In subsequent sections, we first present our approach to tackling the safety verification problem of delayed differential dynamics coupled with one single constant delay (i.e., $k = 1$ in Eq. (1)) in an unbounded time domain, by leveraging a quantitative

stability criterion, if existent, for the linearized counterpart of the potentially nonlinear dynamics in question. A natural extension of this approach to cater for dynamics with multiple delay terms will be remarked thereafter. In what follows, we start the elaboration of the method from DDEs of linear dynamics that admit spectral analysis, and move to nonlinear cases afterwards and show how the linearization technique can be exploited therein.

3 Linear Dynamics

Consider the linear sub-class of dynamics given in Eq. (1):

$$\begin{cases} \dot{\mathbf{x}}(t) = A\mathbf{x}(t) + B\mathbf{x}(t-r), & t \in [0, \infty) \\ \mathbf{x}(t) = \phi(t), & t \in [-r, 0] \end{cases} \quad (3)$$

where $A, B \in \mathbb{R}^{n \times n}$, $\phi \in \mathcal{C}_r$, and the system is associated with the *characteristic equation*

$$\det(zI - A - Be^{-rz}) = 0, \quad (4)$$

where I is the $n \times n$ identity matrix. Denote by $h(z) \doteq zI - A - Be^{-rz}$ the *characteristic matrix* in the sequel. Notice that the characteristic equation can be obtained by seeking nontrivial solutions to Eq. (3) of the form $\xi_\phi(t) = ce^{zt}$, where c is an n -dimensional nonzero constant vector.

The roots $\lambda \in \mathbb{C}$ of Eq. (4) are called *characteristic roots* or *eigenvalues* and the set of all eigenvalues is referred to as the *spectrum*, denoted by $\sigma = \{\lambda \mid \det(h(\lambda)) = 0\}$. Due to the exponentiation in the characteristic equation, the DDE has, in line with its infinite-dimensional nature, infinitely many eigenvalues possibly, making a spectral analysis more involved. The spectrum does however enjoy some elementary properties that can be exploited in the analysis. For instance, the spectrum has no finite accumulation point in \mathbb{C} and therefore for each positive $\gamma \in \mathbb{R}$, the number of roots satisfying $|\lambda| \leq \gamma$ is finite. It follows that the spectrum is a countable (albeit possibly infinite) set:

Lemma 1 (Accumulation freedom [6,19]). *Given $\gamma \in \mathbb{R}$, there are at most finitely many characteristic roots satisfying $\Re(\lambda) > \gamma$. If there is a sequence $\{\lambda_n\}$ of roots of Eq. (4) such that $|\lambda_n| \rightarrow \infty$ as $n \rightarrow \infty$, then $\Re(\lambda_n) \rightarrow -\infty$ as $n \rightarrow \infty$.*

Lemma 1 suggests that there are only a finite number of solutions in any vertical strip in the complex plane, and there thus exists an upper bound $\alpha \in \mathbb{R}$ such that every characteristic root λ in the spectrum satisfies $\Re(\lambda) < \alpha$. This upper bound captures essentially the asymptotic behavior of the linear dynamics:

Theorem 1 (Globally exponential stability [6,36]). *Suppose $\Re(\lambda) < \alpha$ for every characteristic root λ . Then there exists $K > 0$ such that*

$$\|\xi_\phi(t)\| \leq K \|\phi\| e^{\alpha t}, \quad \forall t \geq 0, \forall \phi \in \mathcal{C}_r, \quad (5)$$

where $\xi_\phi(t)$ is the solution to Eq. (3). In particular, $\mathbf{x} = \mathbf{0}$ is a globally exponentially stable equilibrium of Eq. (3) if $\Re(\lambda) < 0$ for every characteristic root; it is unstable if there is a root satisfying $\Re(\lambda) > 0$.

Theorem 1 establishes an existential guarantee that the solution to the linear delayed dynamics approaches the zero equilibrium exponentially for any initial conditions in \mathcal{C}_r . To achieve automatic safety verification, however, we ought to find a constructive means of estimating the (signed) rate of convergence α and the coefficient K in Eq. (5). This motivates the introduction of the so-called *fundamental solution* $\xi_{\phi'}(t)$ to Eq. (3), whose Laplace transform will later be shown to be $h^{-1}(z)$, the inverse characteristic matrix, which always exists for z satisfying $\Re(z) > \max_{\lambda \in \sigma} \Re(\lambda)$.

Lemma 2 (Variation-of-constants [19, 36]). *Let $\xi_{\phi}(t)$ be the solution to Eq. (3). Denote by $\xi_{\phi'}(t)$ the solution that satisfies Eq. (3) for $t \geq 0$ and satisfies a variation of the initial condition as $\phi'(0) = I$ and $\phi'(t) = O$ for all $t \in [-r, 0]$, where O is the $n \times n$ zero matrix, then for $t \geq 0$,*

$$\xi_{\phi}(t) = \xi_{\phi'}(t)\phi(0) + \int_0^t \xi_{\phi'}(t-\tau)B\phi(\tau-r) d\tau. \quad (6)$$

Note that in Eq. (6), $\phi(t)$ is extended to $[-r, \infty)$ by making it zero for $t > 0$. In spite of the discontinuity of ϕ' at zero, the existence of the solution $\xi_{\phi'}(t)$ can be proven by the well-known method of steps [8].

Lemma 3 (Fundamental solution [19]). *The solution $\xi_{\phi'}(t)$ to Eq. (3) with initial data ϕ' is the fundamental solution; that is for z s.t. $\Re(z) > \max_{\lambda \in \sigma} \Re(\lambda)$,*

$$\mathcal{L}\{\xi_{\phi'}\}(z) = h^{-1}(z).$$

The fundamental solution $\xi_{\phi'}(t)$ can be proven to share the same exponential bound as that in Theorem 1, while the following theorem, as a consequence of Lemma 2, gives an exponential estimation of $\xi_{\phi}(t)$ in connection with $\xi_{\phi'}(t)$:

Theorem 2 (Exponential estimation [36]). *Denote by $\mu \equiv \max_{\lambda \in \sigma} \Re(\lambda)$ the maximum real part of eigenvalues in the spectrum. Then for any $\alpha > \mu$, there exists $K > 0$ such that*

$$\|\xi_{\phi'}(t)\| \leq K e^{\alpha t}, \quad \forall t \geq 0, \quad (7)$$

and hence by Eq. (6), $\|\xi_{\phi}(t)\| \leq K (1 + \|B\| \int_0^t e^{-\alpha \tau} d\tau) \|\phi\| e^{\alpha t}$ for any $t \geq 0$ and $\phi \in \mathcal{C}_r$. In particular, $\mathbf{x} = \mathbf{0}$ is globally exponentially stable for Eq. (3) if $\mu < 0$.

Following Theorem 2, an exponentially decreasing bound on the solution $\xi_{\phi}(t)$ to linear DDEs of the form (3) can be assembled by computing α satisfying $\mu < \alpha < 0$ and the coefficient $K > 0$.

3.1 Identifying the Rightmost Roots

Due to the significance of characteristic roots in the context of stability and bifurcation analysis, numerical methods on identifying—particularly the rightmost—roots of linear (or linearized) DDEs have been extensively studied in the past few decades, see e.g., [3, 11, 43, 45]. There are indeed complete methods on isolating real roots of polynomial exponential functions, for instances [37] and [15] based on cylindrical algebraic decomposition (CAD). Nevertheless, as soon as non-trivial exponential functions arise

in the characteristic equation, there appear to be few, if any, symbolic approaches to detecting complex roots of the equation.

In this paper, we find α that bounds the spectrum from the right of the complex plane, by resorting to the numerical approach developed in [11]. The computation therein employs discretization of the solution operator using linear multistep (LMS) methods to approximate eigenvalues of linear DDEs with multiple constant delays, under an absolute error of $\mathcal{O}(\tau^p)$ for sufficiently small stepsize τ , where $\mathcal{O}(\cdot)$ is the big Omicron notation and p depends on the order of the LMS-methods. A well-developed MATLAB package called DDE-BIFTOOL [10] is furthermore available to mechanize the computation, which will be demonstrated in our forthcoming examples.

3.2 Constructing K

By the inverse Laplace transform (cf. Theorem 5.2 in [19] for a detailed proof), we have $\xi_{\phi'}(t) = \lim_{V \rightarrow \infty} \frac{1}{2\pi i} \int_{\alpha-iV}^{\alpha+iV} e^{zt} h^{-1}(z) dz$ for z satisfying $\Re(z) > \mu$, where α is the exponent associated with the bound on $\xi_{\phi'}(t)$ in Eq. (7), and hence by substituting $z = \alpha + i\nu$, we have

$$e^{-\alpha t} \xi_{\phi'}(t) = \lim_{V \rightarrow \infty} \frac{1}{2\pi} \int_{-V}^V e^{i\nu t} h^{-1}(\alpha + i\nu) d\nu.$$

Since $h^{-1}(z) = \frac{I}{z} + (h^{-1}(z) - \frac{I}{z}) = \frac{I}{z} + \mathcal{O}(1/z^2)$, together with the fact that an integral over a quadratic integrand is convergent, it follows that

$$e^{-\alpha t} \xi_{\phi'}(t) = \lim_{V \rightarrow \infty} \frac{1}{2\pi} \int_{-V}^V e^{i\nu t} \frac{I}{\alpha + i\nu} d\nu + \frac{1}{2\pi} \int_{-\infty}^{\infty} e^{i\nu t} \mathcal{O}\left(\frac{1}{(\alpha + i\nu)^2}\right) d\nu.$$

By taking the norm while observing that $|e^{i\nu t}| = 1$, we get

$$e^{-\alpha t} \|\xi_{\phi'}(t)\| \leq \underbrace{\left\| \lim_{V \rightarrow \infty} \frac{1}{2\pi} \int_{-V}^V e^{i\nu t} \frac{I}{\alpha + i\nu} d\nu \right\|}_{(8-a)} + \underbrace{\frac{1}{2\pi} \int_{-\infty}^{\infty} \left\| \mathcal{O}\left(\frac{1}{(\alpha + i\nu)^2}\right) \right\| d\nu}_{(8-b)}. \quad (8)$$

For the integral (8-a), the fact¹ that

$$\int_{-\infty}^{\infty} \frac{e^{iax}}{b+ix} dx = \int_{-\infty}^{\infty} \frac{e^{ix}}{ab+ix} dx = \begin{cases} 2\pi e^{-ab} & \text{if } a, b > 0 \\ 0 & \text{if } a > 0, b < 0, \end{cases} \quad (9)$$

implies

$$\left\| \lim_{V \rightarrow \infty} \frac{1}{2\pi} \int_{-V}^V e^{i\nu t} \frac{I}{\alpha + i\nu} d\nu \right\| \leq \begin{cases} 1, & \forall t > 0, \forall \alpha > 0 \\ 0, & \forall t > 0, \forall \alpha < 0. \end{cases} \quad (10)$$

Notice that the second integral (8-b) is computable, since it is convergent and independent of t . The underlying computation of the *improper integral*, however, can be rather time-consuming. We therefore detour by computing an upper bound of (8-b) in the form of a *definite integral*, due to Lemma 4, which suffices to constitute an exponential estimation of $\xi_{\phi'}(t)$ while reducing computational efforts pertinent to the integration.

¹ The integral in (9) is divergent for $a = 0$ or $b = 0$ in the sense of a Riemann integral.

Lemma 4. *There exists $M > 0$ such that inequation (11) below holds for any $\alpha > \mu$.*

$$\int_{-\infty}^{\infty} \left\| \mathcal{O}\left(\frac{1}{(\alpha + i\nu)^2}\right) \right\| d\nu \leq \int_{-M}^{M} \left\| \mathcal{O}\left(\frac{1}{(\alpha + i\nu)^2}\right) \right\| d\nu + \frac{8n}{M} (\|A\| + \|B\| e^{-r\alpha}) \quad (11)$$

where $\mu \hat{=} \max_{\lambda \in \sigma} \Re(\lambda)$, $z = \alpha + i\nu$, and n is the order of A and B .

Proof. The proof depends essentially on constructing a threshold $M > 0$ such that the integral over $|\nu| > M$ can be bounded, thus transforming the improper integral in question to a definite one. To find such an M , observe that

$$\left\| \mathcal{O}\left(\frac{1}{z^2}\right) \right\| = \left\| h^{-1}(z) - \frac{I}{z} \right\| = \|h^{-1}(z)\| \left\| I - \frac{h(z)}{z} \right\| \leq \frac{\|h^{-1}(z)\|}{|z|} (\|A\| + \|B\| e^{-r\alpha}).$$

Without loss of generality, suppose the entry of $h^{-1}(z)$ at (i, j) takes the form

$$\begin{aligned} (h^{-1})_{ij} &= \left(\sum_{k=0}^{n-1} p_k^{ij}(e^{-rz}) z^k \right) / \det(h(z)) = \left(\sum_{k=0}^{n-1} p_k^{ij}(e^{-rz}) z^k \right) / (z^n + \sum_{k=0}^{n-1} q_k(e^{-rz}) z^k) \\ &= \frac{1}{z} \left(\sum_{k=0}^{n-1} p_k^{ij}(e^{-rz}) z^{k-n+1} \right) / (1 + \sum_{k=0}^{n-1} q_k(e^{-rz}) z^{k-n}), \end{aligned}$$

where $p_k^{ij}(\cdot)$ and $q_k(\cdot)$ are polynomials in e^{-rz} as coefficients of z^k . Since e^{-rz} is bounded by $e^{-r\alpha}$ along the vertical line $z = \alpha + i\nu$, we can conclude that there exist P_k^{ij} and Q_k such that $|p_k^{ij}(e^{-rz})| \leq P_k^{ij}$ and $|q_k(e^{-rz})| \leq Q_k$, with $P_{n-1}^{ij} = 1$ if $i = j$, and 0 otherwise. Furthermore, in the vertical line $z = \alpha + i\nu$, if $|\nu| \geq 1$, then

$$\begin{aligned} \left| \sum_{k=0}^{n-1} p_k^{ij}(e^{-rz}) z^{k-n+1} \right| &\leq |p_{n-1}^{ij}(e^{-rz})| + \sum_{k=0}^{n-2} |p_k^{ij}(e^{-rz}) z^{-1}| \leq P_{n-1}^{ij} + \sum_{k=0}^{n-2} P_k^{ij} |z^{-1}|, \\ \left| 1 + \sum_{k=0}^{n-1} q_k(e^{-rz}) z^{k-n} \right| &\geq 1 - \sum_{k=0}^{n-1} |q_k(e^{-rz})| |z^{k-n}| \geq 1 - \sum_{k=0}^{n-1} Q_k |z^{-1}|. \end{aligned}$$

We can thus choose $|\nu| > M \hat{=} \max_{1 \leq i, j \leq n} \left\{ 1, 2 \sum_{k=0}^{n-1} Q_k, \sum_{k=0}^{n-2} P_k^{ij} \right\}$, which implies

$$\begin{aligned} \left| \left(\sum_{k=0}^{n-1} p_k^{ij}(e^{-rz}) z^k \right) / \det(h(z)) \right| &\leq \left| \frac{1}{z} \left(\sum_{k=0}^{n-1} p_k^{ij}(e^{-rz}) z^{k-n+1} \right) / (1 + \sum_{k=0}^{n-1} q_k(e^{-rz}) z^{k-n}) \right| \\ &\leq \left| \frac{1}{z} \right| (P_{n-1}^{ij} + \sum_{k=0}^{n-2} P_k^{ij} |z^{-1}|) / (1 - \sum_{k=0}^{n-1} Q_k |z^{-1}|) \leq \frac{2}{|z|} (1 + P_{n-1}^{ij}) \leq \frac{4}{|z|}, \end{aligned}$$

where the third inequality holds since $|\nu| > M$. It then follows, if $|\nu| > M$, that

$$\left\| \mathcal{O}\left(\frac{1}{(\alpha + i\nu)^2}\right) \right\| \leq \frac{\|h^{-1}(z)\|}{|z|} (\|A\| + \|B\| e^{-r\alpha}) \leq \frac{4n}{\nu^2} (\|A\| + \|B\| e^{-r\alpha}),$$

and thereby

$$\begin{aligned} \int_{-\infty}^{\infty} \left\| \mathcal{O}\left(\frac{1}{(\alpha + i\nu)^2}\right) \right\| d\nu &\leq \int_{-M}^M \left\| \mathcal{O}\left(\frac{1}{(\alpha + i\nu)^2}\right) \right\| d\nu + 2 \int_M^{\infty} \frac{4n}{\nu^2} (\|A\| + \|B\| e^{-r\alpha}) d\nu \\ &\leq \int_{-M}^M \left\| \mathcal{O}\left(\frac{1}{(\alpha + i\nu)^2}\right) \right\| d\nu + \frac{8n}{M} (\|A\| + \|B\| e^{-r\alpha}). \end{aligned}$$

This completes the proof. \square

Equations (8), (10) and (11) yield that $e^{-\alpha t} \|\xi_{\phi'}(t)\|$ is upper-bounded by

$$K = \frac{1}{2\pi} \left(\int_{-M}^M \left\| \mathcal{O}\left(\frac{1}{(\alpha + i\nu)^2}\right) \right\| d\nu + \frac{8n}{M} (\|A\| + \|B\| e^{-r\alpha}) \right) + 1_0(\alpha), \quad (12)$$

for all $t > 0$. Here M is the constant given in Lemma 4, while $1_0: (\mu, \infty) \setminus \{0\} \mapsto \{0, 1\}$ is an indicator function² of $\{\alpha \mid \alpha > 0\}$, i.e., $1_0(\alpha) = 1$ for $\alpha > 0$ and $1_0(\alpha) = 0$ for $\mu < \alpha < 0$.

In contrast to the existential estimation guarantee established in Theorem 2, exploiting the construction of α and K gives a constructive quantitative criterion permitting to reduce an unbounded safety verification problem to its bounded counterpart:

Theorem 3 (Equivalence of bounded and unbounded safety). *Given $\mathcal{X} \subseteq \mathbb{R}^n$ a set of initial states and $\mathcal{U} \subseteq \mathbb{R}^n$ a set of bad states satisfying $\mathbf{0} \notin \bar{\mathcal{U}}$, suppose we have α satisfying $\mu < \alpha < 0$ and K from Eq. (12). Let $\hat{K} \hat{=} K (1 + \|B\| \int_0^T e^{-\alpha\tau} d\tau) \|\mathcal{X}\|$, then there exists $T^* < \infty$, defined as*

$$T^* \hat{=} \max\{0, \inf\{T \mid \forall t > T: [-\hat{K}e^{\alpha t}, \hat{K}e^{\alpha t}]^n \cap \mathcal{U} = \emptyset\}\}, \quad (13)$$

such that for any $T > T^*$, the system (3) is ∞ -safe iff it is T -safe.

Proof. The “only if” part is for free, as ∞ -safety subsumes by definition T -safety. For the “if” direction, the constructed K in Eq. (12) suffices as an upper bound of $e^{-\alpha t} \|\xi_{\phi'}(t)\|$, and hence by Theorem 2, $\|\xi_{\phi}(t)\| \leq \hat{K}e^{\alpha t}$ for any $t \geq 0$ and ϕ constrained by \mathcal{X} . As a consequence, it suffices to show that T^* given by Eq. (13) is finite, which then by definition implies that system (3) is safe over $t > T^*$. Note that the assumption $\mathbf{0} \notin \bar{\mathcal{U}}$ implies that there exists a ball $B(\mathbf{0}, \delta)$ such that $B(\mathbf{0}, \delta) \cap \mathcal{U} = \emptyset$. Moreover, $\hat{K}e^{\alpha t}$ is strictly monotonically decreasing w.r.t. t , and thus $T = \max\{0, \ln(\delta/\hat{K})/\alpha\}$ is an upper bound³ of T^* , which further implies $T^* < \infty$. \square

Example 2 (PD-controller [17]). Consider a PD-controller with linear dynamics defined, for $t \geq 0$, as

$$\dot{y}(t) = v(t); \quad \dot{v}(t) = -\kappa_p(y(t-r) - y^*) - \kappa_d v(t-r), \quad (14)$$

which controls the position y and velocity v of an autonomous vehicle by adjusting its acceleration according to the current distance to a reference position y^* . A constant time

² We rule out the case of $\alpha = 0$, which renders the integral in Eq. (12) divergent.

³ Note that the larger δ is, the tighter bound T will be.

delay r is introduced to model the time lag due to sensing, computation, transmission, and/or actuation. We instantiate the parameters following [17] as $\kappa_p = 2$, $\kappa_d = 3$, $y^* = 1$, and $r = 0.35$. The system described by Eq. (14) then has one equilibrium at $(1; 0)$, which shares equivalent stability with the zero equilibrium of the following system, with $\hat{y} = y - 1$ and $\hat{v} = v$:

$$\dot{\hat{y}}(t) = \hat{v}(t); \quad \dot{\hat{v}}(t) = -2\hat{y}(t-r) - 3\hat{v}(t-r). \quad (15)$$

Suppose we are interested in exploiting the safety property of the system (15) in an unbounded time domain, relative to the set of initial states $\mathcal{X} = [-0.1, 0.1] \times [0, 0.1]$ and the set of unsafe states $\mathcal{U} = \{(\hat{y}; \hat{v}) \mid |\hat{y}| > 0.2\}$. Following our construction process, we obtain automatically some key arguments (depicted in Fig. 1) as $\alpha = -0.5$, $M = 11.9125$, $K = 7.59162$ and $\hat{K} = 2.21103$, which consequently yield $T^* = 4.80579$ s. By Theorem 3, the unbounded safety verification problem thus is reduced to a T -bounded one for any $T > T^*$, inasmuch as ∞ -safety is equivalent to T -safety for the underlying dynamics.

$[-\hat{K}e^{\alpha t}, \hat{K}e^{\alpha t}]^n$ in Eq. (13) can be viewed as an overapproximation of all trajectories originating from \mathcal{X} . As shown in the right part of Fig. 1, this overapproximation, however, is obviously too conservative to be utilized in proving or disproving almost any safety specifications of practical interest. The contribution of our approach lies in the reduction of unbounded verification problems to their bounded counterparts, thereby yielding a quantitative time bound T^* that substantially “trims off” the verification efforts pertaining to $t > T^*$. The derived T -safety verification task can be tackled effectively by methods dedicated to bounded verification of DDEs of the form (3), or more generally, (1), e.g., approaches in [17] and [4].

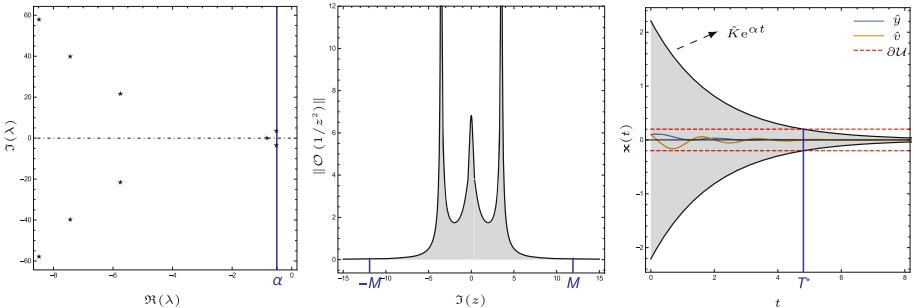


Fig. 1. Left: the identified rightmost roots of $h(z)$ in DDE-BIFTOOL and an upper bound $\alpha = -0.5$ such that $\max_{\lambda \in \sigma} \Re(\lambda) < \alpha < 0$; Center: $M = 11.9125$ that suffices to split and hence upper-bound the improper integral $\int_{-\infty}^{\infty} \|\mathcal{O}(1/z^2)\| d\nu$ in Eq. (11); Right: the obtained time instant $T^* = 4.80579$ s guaranteeing the equivalence of ∞ -safety and T -safety of the PD-controller, for any $T > T^*$.

4 Nonlinear Dynamics

In this section, we address a more general form of dynamics featuring substantial nonlinearity, by resorting to linearization techniques and thereby establishing a quantitative stability criterion, analogous to the linear case, for nonlinear delayed dynamics.

Consider a singly delayed version of Eq. (1):

$$\begin{cases} \dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), \mathbf{x}(t-r)), & t \in [0, \infty) \\ \mathbf{x}(t) = \phi(t), & t \in [-r, 0] \end{cases} \quad (16)$$

with \mathbf{f} being a nonlinear vector field involving possibly non-polynomial functions. Let

$$\mathbf{f}(\mathbf{x}, \mathbf{y}) = A\mathbf{x} + B\mathbf{y} + \mathbf{g}(\mathbf{x}, \mathbf{y}), \text{ with } A = \mathbf{f}_{\mathbf{x}}(\mathbf{0}, \mathbf{0}), B = \mathbf{f}_{\mathbf{y}}(\mathbf{0}, \mathbf{0}),$$

where $\mathbf{f}_{\mathbf{x}}$ and $\mathbf{f}_{\mathbf{y}}$ are the Jacobian matrices of \mathbf{f} in terms of \mathbf{x} and \mathbf{y} , respectively; \mathbf{g} is a vector-valued, high-order term whose Jacobian matrix at $(\mathbf{0}, \mathbf{0})$ is O .

By dropping the high-order term \mathbf{g} in \mathbf{f} , we get the linearized counterpart of Eq. (16):

$$\begin{cases} \dot{\mathbf{x}}(t) = A\mathbf{x}(t) + B\mathbf{x}(t-r), & t \in [0, \infty) \\ \mathbf{x}(t) = \phi(t), & t \in [-r, 0] \end{cases} \quad (17)$$

which falls in the scope of linear dynamics specified in Eq. (3), and therefore is associated with a characteristic equation of the same form as that in Eq. (4). Equation (17) will be in the sequel referred to as the linearization of Eq. (16) at the steady state $\mathbf{0}$, and σ is used to denote the spectrum of the characteristic equation corresponding to Eq. (17).

In light of the well-known Hartman-Grobman theorem [18, 20] in the realm of dynamical systems, the local behavior of a nonlinear dynamical system near a (hyperbolic) equilibrium is qualitatively the same as that of its linearization near this equilibrium. The following statement uncovers the connection between the locally asymptotic behavior of a nonlinear system and the spectrum of its linearization:

Theorem 4 (Locally exponential stability [6, 36]). Suppose $\max_{\lambda \in \sigma} \Re(\lambda) < \alpha < 0$. Then $\mathbf{x} = \mathbf{0}$ is a locally exponentially stable equilibrium of the nonlinear systems (16). In fact, there exists $\delta > 0$ and $K > 0$ such that

$$\|\phi\| \leq \delta \implies \|\xi_{\phi}(t)\| \leq K \|\phi\| e^{\alpha t/2}, \quad \forall t \geq 0,$$

where $\xi_{\phi}(t)$ is the solution to Eq. (16). If $\Re(\lambda) > 0$ for some λ in σ , then $\mathbf{x} = \mathbf{0}$ is unstable.

Akin to the linear case, Theorem 4 establishes an existential guarantee that the solution to the nonlinear delayed dynamics approaches the zero equilibrium exponentially for initial conditions within a δ -neighborhood of this equilibrium. The need of

constructing α , K and δ quantitatively in Theorem 4, as essential to our automatic verification approach, invokes again the fundamental solution $\xi_{\phi'}(t)$ to the linearized dynamics in Eq. (17):

Lemma 5 (Variation-of-constants [19, 36]). *Consider nonhomogeneous systems of the form*

$$\begin{cases} \dot{\mathbf{x}}(t) = A\mathbf{x}(t) + B\mathbf{x}(t-r) + \boldsymbol{\eta}(t), & t \in [0, \infty) \\ \mathbf{x}(t) = \phi(t), & t \in [-r, 0] \end{cases} \quad (18)$$

Let $\xi_\phi(t)$ be the solution to Eq. (18). Denote by $\xi_{\phi'}(t)$ the solution that satisfies Eq. (17) for $t \geq 0$ and satisfies a variation of the initial condition as $\phi'(0) = I$ and $\phi'(t) = O$ for all $t \in [-r, 0)$. Then for $t \geq 0$,

$$\xi_\phi(t) = \xi_{\phi'}(t)\phi(0) + \int_0^t \xi_{\phi'}(t-\tau)B\phi(\tau-r) d\tau + \int_0^t \xi_{\phi'}(t-\tau)\boldsymbol{\eta}(\tau) d\tau, \quad (19)$$

where ϕ is extended to $[-r, \infty)$ with $\phi(t) = 0$ for $t > 0$.

In what follows, we give a constructive quantitative estimation of the solutions to nonlinear dynamics, which admits a reduction of the problem of constructing an exponential upper bound of a nonlinear system to that of its linearization, as being immediately evident from the constructive proof.

Theorem 5 (Exponential estimation). *Suppose that $\max_{\lambda \in \sigma} \Re(\lambda) < \alpha < 0$. Then there exist $K > 0$ and $\delta > 0$ such that $\|\xi_{\phi'}(t)\| \leq K e^{\alpha t}$ for any $t \geq 0$, and*

$$\|\phi\| \leq \delta \implies \|\xi_\phi(t)\| \leq K e^{-r\alpha} \left(1 + \|B\| \int_0^r e^{-\alpha\tau} d\tau \right) \|\phi\| e^{\alpha t/2}, \quad \forall t \geq 0,$$

where $\xi_\phi(t)$ is the solution to nonlinear systems (16) and $\xi_{\phi'}(t)$ is the fundamental solution to the linearized counterpart (17).

Proof. The existence of K follows directly from Eq. (7) in Theorem 2. By the variation-of-constants formula (19), we have, for $t \geq 0$,

$$\xi_\phi(t) = \xi_{\phi'}(t)\phi(0) + \int_0^t \xi_{\phi'}(t-\tau)B\phi(\tau-r) d\tau + \int_0^t \xi_{\phi'}(t-\tau)\mathbf{g}(\mathbf{x}(\tau), \mathbf{x}(\tau-r)) d\tau, \quad (20)$$

where ϕ is extended to $[-r, \infty)$ with $\phi(t) = 0$ for $t > 0$. Define $\mathbf{x}_t^\phi(\cdot) \in \mathcal{C}_r$ as $\mathbf{x}_t^\phi(\theta) = \xi_\phi(t+\theta)$ for $\theta \in [-r, 0]$. Then $\mathbf{g}(\cdot, \cdot)$ being a higher-order term yields that for any $\epsilon > 0$, there exists $\delta_\epsilon > 0$ such that $\|\mathbf{x}_t^\phi\| \leq \delta_\epsilon$ implies $\mathbf{g}(\mathbf{x}(t), \mathbf{x}(t-r)) \leq \epsilon \|\mathbf{x}_t^\phi\|$. Due to the fact that $\|\xi_{\phi'}(t)\| \leq K e^{\alpha t}$ and the monotonicity of $\|\xi_{\phi'}(t)\|$ with $\alpha < 0$, we have $\|\mathbf{x}_t^{\phi'}\| \leq K e^{\alpha(t-r)}$. This, together with Eq. (20), leads to

$$\begin{aligned} \|\mathbf{x}_t^\phi\| &\leq K \|\phi\| e^{\alpha(t-r)} + \int_0^r K \|B\| \|\phi\| e^{\alpha(t-r)} e^{-\alpha\tau} d\tau + \int_0^t K e^{\alpha(t-r)} e^{-\alpha\tau} \|\mathbf{x}_\tau^\phi\| d\tau \\ &= K \left(1 + \|B\| \int_0^r e^{-\alpha\tau} d\tau \right) \|\phi\| e^{\alpha(t-r)} + \epsilon K e^{\alpha(t-r)} \int_0^t e^{-\alpha\tau} \|\mathbf{x}_\tau^\phi\| d\tau. \end{aligned}$$

Hence,

$$e^{-\alpha t} \|\mathbf{x}_t^\phi\| \leq K e^{-r\alpha} \left(1 + \|B\| \int_0^r e^{-\alpha\tau} d\tau \right) \|\phi\| + \epsilon K e^{-r\alpha} \int_0^t e^{-\alpha\tau} \|\mathbf{x}_\tau^\phi\| d\tau.$$

By the Grönwall-Bellman inequality [1] we obtain

$$e^{-\alpha t} \|\mathbf{x}_t^\phi\| \leq K e^{-r\alpha} \left(1 + \|B\| \int_0^r e^{-\alpha\tau} d\tau \right) \|\phi\| e^{\epsilon K e^{-r\alpha} t}$$

and thus

$$\|\mathbf{x}_t^\phi\| \leq K e^{-r\alpha} \left(1 + \|B\| \int_0^r e^{-\alpha\tau} d\tau \right) \|\phi\| e^{\epsilon K e^{-r\alpha} t + \alpha t}.$$

Set $\epsilon \leq -\alpha/(2K e^{-r\alpha})$ and $\delta = \min \{ \delta_\epsilon, \delta_\epsilon / (K e^{-r\alpha} (1 + \|B\| \int_0^r e^{-\alpha\tau} d\tau)) \}$. This yields, for any $t \geq 0$,

$$\|\phi\| \leq \delta \implies \|\xi_\phi(t)\| \leq K e^{-r\alpha} \left(1 + \|B\| \int_0^r e^{-\alpha\tau} d\tau \right) \|\phi\| e^{\alpha t/2},$$

completing the proof. \square

The above constructive quantitative estimation of the solutions to nonlinear dynamics gives rise to the reduction, analogous to the linear case, of unbounded verification problems to bounded ones, in the presence of a local stability criterion.

Theorem 6 (Equivalence of safety properties). *Given initial state set $\mathcal{X} \subseteq \mathbb{R}^n$ and bad states $\mathcal{U} \subseteq \mathbb{R}^n$ satisfying $\mathbf{0} \notin \bar{\mathcal{U}}$. Let σ denote the spectrum of the characteristic equation corresponding to Eq. (17). Suppose that $\max_{\lambda \in \sigma} \Re(\lambda) < \alpha < 0$, and the fundamental solution to Eq. (17) satisfies $\|\xi_\phi(t)\| \leq K e^{\alpha t}$ for any $t \geq 0$. Let $\tilde{K} = K e^{-r\alpha} (1 + \|B\| \int_0^r e^{-\alpha\tau} d\tau) \|\mathcal{X}\|$. Then there exists $\delta > 0$ and $T^* < \infty$, defined as*

$$T^* \triangleq \max\{0, \inf\{T \mid \forall t > T: [-\tilde{K} e^{\alpha t/2}, \tilde{K} e^{\alpha t/2}]^n \cap \mathcal{U} = \emptyset\}\},$$

such that if $\|\mathcal{X}\| \leq \delta$, then for any $T > T^*$, the system (16) is ∞ -safe iff it is T -safe.

Proof. The proof is analogous to that of Theorem 3, particularly following from the local stability property stated in Theorem 5. \square

Note that for nonlinear dynamics, the equivalence of safety claimed by Theorem 6 holds on the condition that $\|\mathcal{X}\| \leq \delta$, due to the locality stemming from linearization. In fact, such a set $\mathfrak{B} \subseteq \mathbb{R}^n$ satisfying $\|\mathfrak{B}\| \leq \delta$ describes (a subset of) the basin of attraction around the local attractor $\mathbf{0}$, in a sense that any initial condition in \mathfrak{B} will lead the trajectory eventually into the attractor. Consequently, for verification problems where $\mathcal{X} \supseteq \mathfrak{B}$, if the reachable set originating from \mathcal{X} is guaranteed to be subsumed within \mathfrak{B} in the time interval $[T' - r, T']$, then $T' + T^*$ suffices as a bound to avoid unbounded verification, namely for any $T > T' + T^*$, the system is ∞ -safe iff it is T -safe. This is furthermore demonstrated by the following example.

Example 3 (Population dynamics [4, 25]). Consider a slightly modified version of the delayed logistic equation introduced by G. Hutchinson in 1948 (cf. [22])

$$\dot{N}(t) = N(t)[1 - N(t - r)], \quad t \geq 0, \quad (21)$$

which is used to model a single population whose per capita rate of growth $\dot{N}(t)/N(t)$ depends on the population size r time units in the past. This would be a reasonable model for a population that features a significant minimum reproductive age or depends on a resource, like food, needing time to grow and thus to recover its availability.

If we change variables, putting $u = N - 1$, then Eq. (21) becomes the famous Wright's equation (see [44]):

$$\dot{u}(t) = -u(t - r)[1 + u(t)], \quad t \geq 0. \quad (22)$$

The steady state $N = 1$ is now $u = 0$. We instantiate the verification problem of Eq. (22) over $[-r, \infty)$ as $\mathcal{X} = [-0.2, 0.2]$, $\mathcal{U} = \{u \mid |u| > 0.6\}$, under a constant delay $r = 1$. Note that delay-independent Lyapunov techniques, e.g. [32], cannot solve this problem, since Wright's conjecture [44], which has been recently proven in [40], together with corollaries thereof implies that there does not exist a Lyapunov functional guaranteeing absolute stability of Eq. (22) with arbitrary constant delays. To achieve an exponential estimation, we first linearize the dynamics by dropping the nonlinearity $u(t)u(t - r)$ thereof:

$$\dot{v}(t) = -v(t - 1), \quad t \geq 0. \quad (23)$$

Following our constructive approach, we obtain automatically for Eq. (23) $\alpha = -0.3$ (see the left of Fig. 2), $M = 2.69972$, $K = 3.28727$, and thereby for Eq. (22) $\delta = 0.00351678$, $\tilde{K} = 0.0338039$ and $T^* = 0$ s. It is worth highlighting that by the bounded verification method in [17], with Taylor models of the order 5, an overapproximation Ω of the reachable set w.r.t. system (22) over the time interval $[14.5, 15.5]$ was verified to be enclosed in the δ -neighborhood of 0 , i.e., $\|\Omega\| \leq \delta$, yet escaped from this region around $t = 55.3$ s, and tended to diverge soon, as depicted in the right part of Fig. 2, and thus cannot prove unbounded safety properties. However, with our result of $T^* = 0$ s and the fact that Ω over $[-1, 15.5]$ is disjoint with \mathcal{U} , we are able to claim safety of the underlying system over an infinite time domain.

DDEs with Multiple Different Delays. Delay differential equations with multiple fixed discrete delays are extensively used in the literature to model practical systems where components coupled with different time lags coexist and interact with each other. We remark that previous theorems on exponential estimation and equivalence of safety w.r.t. cases of single delay extend immediately to systems of the form (1) with almost no change, except for replacing $\|B\| e^{-r\alpha}$ with $\sum_{i=1}^k \|A_i\| e^{-r_i\alpha}$ and $\|B\|$ with $\sum_{i=1}^k \|A_i\|$, where A_i denotes the matrix attached to $\mathbf{x}(t - r_i)$ in the linearization. For a slightly modified form of the variation-of-constants formula under multiple delays, we refer the readers to Theorem 1.2 in [19].

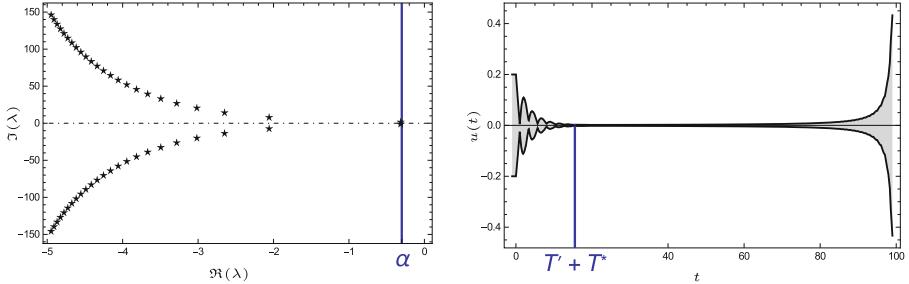


Fig. 2. Left: the identified rightmost eigenvalues of $h(z)$ and an upper bound $\alpha = -0.5$ such that $\max_{\lambda \in \sigma} \Re(\lambda) < \alpha < 0$; Right: overapproximation of the reachable set of the system (22) produced by the method in [17] using Taylor models for bounded verification. Together with this overapproximation we prove the equivalence of ∞ -safety and T -safety of the system, for any $T > (T' + T^*) = 15.5$ s.

5 Implementation and Experimental Results

To further investigate the scalability and efficiency of our constructive approach, we have carried out a prototypical implementation⁴ in Wolfram MATHEMATICA, which was selected due to its built-in primitives for integration and matrix operations. By interfacing with DDE-BIFTOOL⁵ (in MATLAB or GNU OCTAVE) for identifying the rightmost characteristic roots of linear (or linearized) DDEs, our implementation computes an appropriate T^* that admits a reduction of unbounded verification problems to bounded ones. A set of benchmark examples from the literature has been evaluated on a 3.6 GHz Intel Core-i7 processor with 8 GB RAM running 64-bit Ubuntu 16.04. All computations of T^* were safely rounded and finished within 6 s for any of the examples, including Examples 2 and 3. In what follows, we demonstrate in particular the applicability of our technique to DDEs featuring non-polynomial dynamics, high dimensionality and multiple delays.

Example 4 (Disease pathology [25,27,32]). Consider the following non-polynomial DDE for $t \geq 0$:

$$\dot{p}(t) = \frac{\beta \theta^n p(t-r)}{\theta^n + p^n(t-r)} - \gamma p(t), \quad (24)$$

where $p(t)$ is positive and indicates the number of mature blood cells in circulation, while r models the delay between cell production and cell maturation. We consider the case $\theta = 1$ as in [32]. Constants are instantiated as $n = 1$, $\beta = 0.5$, $\gamma = 0.6$ and $r = 0.5$. The unbounded verification problem of Eq. (24) over $[-r, \infty)$ is configured as $\mathcal{X} = [0, 0.2]$ and $\mathcal{U} = \{p \mid |p| > 0.3\}$. Then the linearization of Eq. (24) reads

$$\dot{p}(t) = -0.6p(t) + 0.5p(t-0.5). \quad (25)$$

⁴ <http://lcs.ios.ac.cn/~chenms/tools/UDDER.tar.bz2>.

⁵ <http://ddebfotool.sourceforge.net/>.

With $\alpha = -0.07$ obtained from DDE-BIFTOOL, our implementation produces for Eq. (25) the values $M = 2.23562$, $K = 1.75081$, and thereby for Eq. (24) $\delta = 0.0163426$, $\tilde{K} = 0.0371712$ and $T^* = 0$ s. Thereafter by the bounded verification method in [17], with Taylor models of the order 5, an overapproximation of the reachable set w.r.t. system (24) over the time interval $[25.45, 25.95]$ was verified to be enclosed in the δ -neighborhood of $\mathbf{0}$. This fact, together with $T^* = 0$ s and the overapproximation on $[-0.5, 25.95]$ being disjoint with \mathcal{U} , yields safety of the system (24) over $[-0.5, \infty)$.

Example 5 (Gene regulation [12,36]). To examine the scalability of our technique to higher dimensions, we recall an instantiation of Eq. (2) by setting $n = 5$, namely with 5 state components $\mathbf{x} = (x_1; \dots; x_5)$ and 5 delay terms $\mathbf{r} = (0.1; 0.2; 0.4; 0.8; 1.6)$ involved, $g(x) = -x$, $\beta_j = 1$ for $j = 1, \dots, 5$, $\mathcal{X} = \mathcal{B}((1; 1; 1; 1; 1), 0.2)$ and $\mathcal{U} = \{\mathbf{x} \mid |x_1| > 1.5\}$. With $\alpha = -0.04$ derived from DDE-BIFTOOL, our implementation returns $M = 64.264$, $K = 4.42207$, $\tilde{K} = 49.1463$ and $T^* = 87.2334$ s, thereby yielding the equivalence of ∞ -safety to T -safety for any $T > T^*$. Furthermore, the safety guarantee issued by the bounded verification method in [4] based on rigorous simulations under $T = 88$ s suffices to prove safety of the system over an infinite time horizon.

6 Conclusion

We have presented a constructive method, based on linearization and spectral analysis, for computing a delay-dependent, exponentially decreasing upper bound, if existent, that encloses trajectories of a DDE originating from a certain set of initial functions. We showed that such an enclosure facilitates a reduction of the verification problem over an unbounded temporal horizon to a bounded one. Preliminary experimental results on a set of representative benchmarks from the literature demonstrate that our technique effectively extends the scope of existing bounded verification techniques to unbounded verification tasks.

Peeking into future directions, we plan to exploit a tight integration of our technique into several automatic tools dedicated to bounded verification of DDEs, as well as more permissive forms of stabilities, e.g. asymptotical stability, that may admit a similar reduction-based idea. An extension of our method to deal with more general forms of DDEs, e.g., with time-varying, or distributed (i.e., a weighted average of) delays, will also be of interest. Additionally, we expect to refine our enclosure of system trajectories by resorting to a topologically finite partition of the initial set of functions.

References

1. Bellman, R.: The stability of solutions of linear differential equations. Duke Math. J. **10**(4), 643–647 (1943)
2. Bellman, R.E., Cooke, K.L.: Differential-difference equations. Technical Report R-374-PR, RAND Corporation, Santa Monica, California, January 1963
3. Breda, D., Maset, S., Vermiglio, R.: Computing the characteristic roots for delay differential equations. IMA J. Numer. Anal. **24**(1), 1–19 (2004)

4. Chen, M., Fränzle, M., Li, Y., Mosaad, P.N., Zhan, N.: Validated simulation-based verification of delayed differential dynamics. In: Fitzgerald, J., Heitmeyer, C., Gnesi, S., Philippou, A. (eds.) FM 2016. LNCS, vol. 9995, pp. 137–154. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48989-6_9
5. Cooke, K.L.: Stability analysis for a vector disease model. *Rocky Mt. J. Math.* **9**(1), 31–42 (1979)
6. Diekmann, O., van Gils, S., Lunel, S., Walther, H.: Delay Equations: Functional-, Complex-, and Nonlinear Analysis. Applied Mathematical Sciences. Springer, New York (2012). <https://doi.org/10.1007/978-1-4612-4206-2>
7. Donzé, A., Maler, O.: Systematic simulation using sensitivity analysis. In: Bemporad, A., Bicchi, A., Buttazzo, G. (eds.) HSCC 2007. LNCS, vol. 4416, pp. 174–189. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71493-4_16
8. Driver, R.: Ordinary and Delay Differential Equations. Applied Mathematical Sciences. Springer, New York (1977). <https://doi.org/10.1007/978-1-4684-9467-9>
9. Duggirala, P.S., Mitra, S., Viswanathan, M.: Verification of annotated models from executions. In: EMSOFT 2013, pp. 26:1–26:10 (2013)
10. Engelborghs, K., Luzyanina, T., Roose, D.: Numerical bifurcation analysis of delay differential equations using DDE-BIFTOOL. *ACM Trans. Math. Softw.* **28**(1), 1–21 (2002)
11. Engelborghs, K., Roose, D.: On stability of LMS methods and characteristic roots of delay differential equations. *SIAM J. Numer. Anal.* **40**(2), 629–650 (2002)
12. Fall, C.P., Marland, E.S., Wagner, J.M., Tyson, J.J. (eds.): Computational Cell Biology, vol. 20. Springer, New York (2002)
13. Fort, J., Méndez, V.: Time-delayed theory of the neolithic transition in Europe. *Phys. Rev. Lett.* **82**(4), 867 (1999)
14. Fränzle, M., Chen, M., Kröger, P.: In memory of Oded Maler: automatic reachability analysis of hybrid-state automata. *ACM SIGLOG News* **6**(1), 19–39 (2019)
15. Gan, T., Chen, M., Li, Y., Xia, B., Zhan, N.: Reachability analysis for solvable dynamical systems. *IEEE Trans. Automat. Contr.* **63**(7), 2003–2018 (2018)
16. Girard, A., Pappas, G.J.: Approximate bisimulation: a bridge between computer science and control theory. *Eur. J. Control* **17**(5–6), 568–578 (2011)
17. Goubault, E., Putot, S., Sahlmann, L.: Inner and outer approximating flowpipes for delay differential equations. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10982, pp. 523–541. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96142-2_31
18. Grobman, D.M.: Homeomorphism of systems of differential equations. *Doklady Akademii Nauk SSSR* **128**(5), 880–881 (1959)
19. Hale, J., Lunel, S.: Introduction to Functional Differential Equations. Applied Mathematical Sciences. Springer, New York (1993). <https://doi.org/10.1007/978-1-4612-4342-7>
20. Hartman, P.: A lemma in the theory of structural stability of differential equations. *Proc. Am. Math. Soc.* **11**(4), 610–620 (1960)
21. Huang, Z., Fan, C., Mitra, S.: Bounded invariant verification for time-delayed nonlinear networked dynamical systems. *Nonlinear Anal. Hybrid Syst.* **23**, 211–229 (2017)
22. Hutchinson, G.E.: Circular causal systems in ecology. *Ann. N. Y. Acad. Sci.* **50**(4), 221–246 (1948)
23. Ikeda, K., Matsumoto, K.: High-dimensional chaotic behavior in systems with time-delayed feedback. *Phys. D Nonlinear Phenom.* **29**(1–2), 223–235 (1987)
24. Krasovskii, N.: Stability of Motion: Applications of Lyapunov's Second Method to Differential Systems and Equations with Delay. Studies in Mathematical Analysis and Related Topics. Stanford University Press, Stanford (1963)
25. Kuang, Y.: Delay Differential Equations: With Applications in Population Dynamics. Mathematics in Science and Engineering. Elsevier Science, Amsterdam (1993)

26. Levine, W.S.: The Control Handbook: Control System Fundamentals. Electrical Engineering Handbook, 2nd edn. CRC Press, Boca Raton (2010)
27. Mackey, M.C., Glass, L.: Oscillation and chaos in physiological control systems. *Science* **197**(4300), 287–289 (1977)
28. Mallet-Paret, J., Sell, G.R.: The Poincaré-Bendixson theorem for monotone cyclic feedback systems with delay. *J. Differ. Equ.* **125**, 441–489 (1996)
29. Nazier Mosaad, P., Fränzle, M., Xue, B.: Temporal logic verification for delay differential equations. In: Sampaio, A., Wang, F. (eds.) ICTAC 2016. LNCS, vol. 9965, pp. 405–421. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46750-4_23
30. Myshkis, A.D.: Lineare Differentialgleichungen mit nachteilendem Argument, vol. 17. VEB Deutscher Verlag der Wissenschaften (1955)
31. Nahhal, T., Dang, T.: Test coverage for continuous and hybrid systems. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 449–462. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73368-3_47
32. Peet, M., Lall, S.: Constructing Lyapunov functions for nonlinear delay-differential equations using semidefinite programming. In: Proceedings of NOLCOS, pp. 381–385 (2004)
33. Pola, G., Pepe, P., Benedetto, M.D.D.: Symbolic models for time-varying time-delay systems via alternating approximate bisimulation. *Int. J. Robust Nonlinear Control* **25**, 2328–2347 (2015)
34. Pola, G., Pepe, P., Benedetto, M.D.D., Tabuada, P.: Symbolic models for nonlinear time-delay systems using approximate bisimulations. *Syst. Control Lett.* **59**(6), 365–373 (2010)
35. Prajna, S., Jadbabaie, A.: Methods for safety verification of time-delay systems. In: CDC 2005, pp. 4348–4353 (2005)
36. Smith, H.: An Introduction to Delay Differential Equations with Applications to the Life Sciences, vol. 57. Springer, New York (2011). <https://doi.org/10.1007/978-1-4419-7646-8>
37. Strzeboński, A.: Cylindrical decomposition for systems transcendental in the first variable. *J. Symb. Comput.* **46**(11), 1284–1290 (2011)
38. Szydłowski, M., Krawiec, A., Tobała, J.: Nonlinear oscillations in business cycle model with time lags. *Chaos Solitons Fractals* **12**(3), 505–517 (2001)
39. Vajta, M.: Some remarks on padé-approximations. In: Proceedings of the 3rd TEMPUS-INTCOM Symposium, vol. 242 (2000)
40. van den Berg, J.B., Jaquette, J.: A proof of Wright's conjecture. *J. Differ. Equ.* **264**(12), 7412–7462 (2018)
41. Volterra, V.: Une théorie mathématique de la lutte pour la vie (1927)
42. Volterra, V.: Sur la théorie mathématique des phénomènes héréditaires. *Journal de mathématiques pures et appliquées* **7**, 249–298 (1928)
43. Vyhlídal, T.: Analysis and synthesis of time delay system spectrum. Ph.D. dissertation, Czech Technical University in Prague (2003)
44. Wright, E.M.: A non-linear difference-differential equation. *J. Reine Angew. Math.* **66–87**, 1955 (1955)
45. Wulf, V., Ford, N.J.: Numerical hopf bifurcation for a class of delay differential equations. *J. Comput. Appl. Math.* **115**(1–2), 601–616 (2000)
46. Xue, B., Mosaad, P.N., Fränzle, M., Chen, M., Li, Y., Zhan, N.: Safe over- and under-approximation of reachable sets for delay differential equations. In: Abate, A., Geeraerts, G. (eds.) FORMATS 2017. LNCS, vol. 10419, pp. 281–299. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-65765-3_16
47. Zou, L., Fränzle, M., Zhan, N., Mosaad, P.N.: Automatic verification of stability and safety for delay differential equations. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9207, pp. 338–355. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21668-3_20

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.



Author Index

- Albarghouthi, Aws I-278
André, Étienne I-520
Arcaini, Paolo I-401
Arcak, Murat I-591
Arechiga, Nikos II-137
Ashok, Pranav I-497
Avni, Guy I-630
- Backes, John II-231
Bansal, Suguman I-60
Barbosa, Haniel II-74
Barrett, Clark I-443, II-23, II-74, II-116
Bayless, Sam II-231
Becker, Heiko II-155
Beckett, Ryan II-305
Beillahi, Sidi Mohamed II-286
Berkovits, Idan II-245
Biswas, Ranadeep II-324
Bloem, Roderick I-630
Bouajjani, Ahmed II-267, II-286
Brain, Martin II-116
Breck, Jason I-335
Busatto-Gaston, Damien I-572
- Černý, Pavol I-140
Češka, Milan I-475
Chatterjee, Krishnendu I-630
Chen, Mingshuai I-650
Cimatti, Alessandro I-376
Coenen, Norine I-121
Cook, Byron II-231
Cyphert, John I-335
- D'Antoni, Loris I-3, I-278, I-335
Damian, Andrei II-344
Darulova, Eva II-155, II-174
Davis, Jennifer A. I-366
Deshmukh, Jyotirmoy II-137
Dill, David L. I-443
Dimitrova, Rayna I-241
Dodge, Catherine II-231
Drăgoi, Cezara II-344
- Dreossi, Tommaso I-432
Drews, Samuel I-278
- Elfar, Mahmoud I-180
Emmi, Michael II-324, II-534
Enea, Constantin II-267, II-286, II-324, II-534
Ernst, Gidon II-208
Erradi, Mohammed II-267
- Farzan, Azadeh I-200
Faymonville, Peter I-421
Fedynkovich, Grigory I-259
Feldman, Yotam M. Y. II-405
Feng, Shenghua I-650
Ferreira, Tiago I-3
Finkbeiner, Bernd I-121, I-241, I-421, I-609
Fränzle, Martin I-650
Fremont, Daniel J. I-432
Frohn, Florian II-426
Furbach, Florian I-355
- Gacek, Andrew II-231
Ganesh, Vijay II-367
Gao, Sicun II-137
García Soto, Miriam I-297
Gastin, Paul I-41
Gavrilenko, Natalia I-355
Ghosh, Shromona I-432
Giannarakis, Nick II-305
Giesl, Jürgen II-426
Gomes, Victor B. F. I-387
Griggio, Alberto I-376
Guo, Xiaojie II-496
Gupta, Aarti I-259
Gurfinkel, Arie I-161, II-367
- Hasuo, Ichiro I-401, I-520
Heljanko, Keijo I-355
Henzinger, Thomas A. I-297, I-630
Hong, Chih-Duo I-455
Hu, Alan J. II-231
Hu, Qinheping I-335

- Huang, Derek A. I-443
 Humphrey, Laura R. I-366
 Hur, Chung-Kil II-445
- Ibeling, Duligur I-443
 Iosif, Radu II-43
- Jagannathan, Suresh II-459
 Jain, Mitesh I-553
 Jonáš, Martin II-64
 Julian, Kyle I-443
- Kahsai, Temesghen II-231
 Kang, Eunsuk I-219
 Kapinski, James II-137
 Katz, Guy I-443
 Kim, Edward I-432
 Kim, Eric S. I-591
 Kincaid, Zachary II-97
 Kingston, Derek B. I-366
 Klein, Felix I-609
 Kochenderfer, Mykel J. I-443
 Kocik, Bill II-231
 Kölbl, Martin I-79
 Kong, Soonho II-137
 Könighofer, Bettina I-630
 Kotelnikov, Evgenii II-231
 Křetínský, Jan I-475, I-497
 Kukovec, Jure II-231
- Lafortune, Stéphane I-219
 Lal, Akash II-386
 Lange, Julien I-97
 Lau, Stella I-387
 Lazarus, Christopher I-443
 Lazić, Marijana II-245
 Lee, Juneyoung II-445
 Lesourd, Maxime II-496
 Leue, Stefan I-79
 Li, Jianwen II-3
 Li, Yangjia II-187
 Lim, Rachel I-443
 Lin, Anthony W. I-455
 Liu, Junyi II-187
 Liu, Mengqi II-496
 Liu, Peizun II-386
 Liu, Tao II-187
 Lopes, Nuno P. II-445
 Losa, Giuliano II-245
- Madhukar, Kumar I-259
 Madsen, Curtis I-540
 Magnago, Enrico I-376
 Mahajan, Ratul II-305
 Majumdar, Rupak I-455
 Manolios, Panagiotis I-553
 Markey, Nicolas I-22
 McLaughlin, Sean II-231
 Memarian, Kayvan I-387
 Meyer, Roland I-355
 Militaru, Alexandru II-344
 Millstein, Todd I-315
 Monmege, Benjamin I-572
 Mukherjee, Sayan I-41
 Murray, Toby II-208
 Myers, Chris J. I-540
 Myreen, Magnus O. II-155
- Nagar, Kartik II-459
 Neupane, Thakur I-540
 Niemetz, Aina II-116
 Nori, Aditya I-315
 Nötzli, Andres II-23, II-74
- Padhi, Saswat I-315
 Padon, Oded II-245
 Pajic, Miroslav I-180
 Pichon-Pharabod, Jean I-387
 Piskac, Ruzica I-609
 Ponce-de-León, Hernán I-355
 Prabhu, Sumanth I-259
 Pranger, Stefan I-630
 Preiner, Mathias II-116
- Rabe, Markus N. II-84
 Ravanbakhsh, Hadi I-432
 Reed, Jason II-231
 Reps, Thomas I-335
 Reynier, Pierre-Alain I-572
 Reynolds, Andrew II-23, II-74, II-116
 Rieg, Lionel II-496
 Roohi, Nima II-137
 Roussanaly, Victor I-22
 Roveri, Marco I-376
 Rozier, Kristin Y. II-3
 Rümmer, Philipp I-455
 Rungta, Neha II-231

- Sagiv, Mooly II-405
 Sammartino, Matteo I-3
 Sanán, David II-515
 Sánchez, César I-121
 Sankur, Ocan I-22, I-572
 Santolucito, Mark I-609
 Schilling, Christian I-297
 Schledjewski, Malte I-421
 Schwenger, Maximilian I-421
 Seshia, Sanjit A. I-432, I-591
 Sewell, Peter I-387
 Shah, Parth I-443
 Shao, Zhong II-496
 Sharma, Rahul I-315
 Shemer, Ron I-161
 Shoham, Sharon I-161, II-245, II-405
 Siegel, Stephen F. II-478
 Silva, Alexandra I-3
 Silverman, Jake II-97
 Sizemore, John II-231
 Solar-Lezama, Armando II-137
 Srinivasan, Preethi II-231
 Srivathsan, B. I-41
 Stalzer, Mark II-231
 Stenger, Marvin I-421
 Strejček, Jan II-64
 Subotić, Pavle II-231
- Tatlock, Zachary II-155
 Tentrup, Leander I-121, I-421
 Thakoor, Shantanu I-443
 Tinelli, Cesare II-23, II-74, II-116
 Tizpaz-Niari, Saeid I-140
 Tonetta, Stefano I-376
 Torfah, Hazem I-241, I-421
 Tripakis, Stavros I-219
 Trivedi, Ashutosh I-140
- Vandikas, Anthony I-200
 Vardi, Moshe Y. I-60, II-3
 Varming, Carsten II-231
 Vazquez-Chanlatte, Marcell I-432
 Vediramana Krishnan, Hari Govind II-367
 Vizel, Yakir I-161, II-367
 Volkova, Anastasia II-174
- Waga, Masaki I-520
 Wahl, Thomas II-386
 Walker, David II-305
 Wang, Shuling II-187
 Wang, Yu I-180
 Weininger, Maximilian I-497
 Whaley, Blake II-231
 Widder, Josef II-344
 Wies, Thomas I-79
 Wilcox, James R. II-405
 Wu, Haoze I-443
- Xu, Xiao II-43
 Xue, Bai I-650
- Ying, Mingsheng II-187
 Ying, Shenggang II-187
 Yoshida, Nobuko I-97
- Zeleznik, Luka I-297
 Zeljić, Aleksandar I-443
 Zennou, Rachid II-267
 Zhan, Bohua II-187
 Zhan, Naijun I-650, II-187
 Zhang, Zhen I-540
 Zhang, Zhenya I-401
 Zhao, Yongwang II-515
 Zheng, Hao I-540